Jeff Forristal

The Challenges of Threat Modeling Modern Applications
Table of Contents
The Challenges of Threat Modeling  
Modern Applications

The process of decomposing an application or computing system into components, in order to address areas of cybersecurity weakness or potential cyber threat, is commonly referred to as threat modeling. Secure software development lifecycles (SDL/SSDL) and other formal application security programs often include threat modeling in the program activity portfolio, making threat modeling a required exercise as part of application development. The Building Security In Maturing Model (BSIMM) measures multiple areas of organizational engagement in architectural analysis (AA category) and attack modeling (AM category). The Identify function in the NIST Cybersecurity Framework includes a risk assessment category (ID.RA) encompassing subcategory goals typically achieved via threat modeling and threat analysis.

Threat modeling approaches can vary from simple to thorough, but all tend to seek the same goal: perform a systematic analysis, from an attacker’s perspective, of viable threats/attacks against your systems, applications, and assets. The output from successful threat modeling can drive defensive prioritizations, architectural requirements, and cybersecurity countermeasures. Bad threat models, on the other hand, can waste time, cause misplaced cybersecurity investment, and lead to overconfidence in security posture. The goal is to right-size the investment in threat modeling to produce meaningful and actionable results by development and operations teams, that also satisfy the organization’s risk management needs (KPIs/OKRs, governance, etc.).

Threat modeling is not a new concept. First generation publications in the 1990s showcased various methodologies using decision/threat/attack trees. Microsoft, as part of their Trustworthy Computing initiative in the 2000s, made large contributions to threat modeling including STRIDE, publishing a seminal book on their internal methodology (Snyder & Swiderski in 2004), refining the use of Data Flow Diagrams (DFDs) for threat modeling purposes, and releasing a free threat modeling tool. There have since been other methodologies (OCTAVE, TRIKE, PASTA, TARA/TAL, etc.) and books released featuring methodology refreshes (Shostack in 2014), plus the introduction of Agile-friendly options like VAST and use of Process Flow Diagrams (PFDs). Threat-centric catalogs (CAPEC, ATT&CK) of adversarial tactics, techniques, and procedures (TTPs) have been codified.
With decades of maturity and refinement, a lot has been learned, collected, and published about threat modeling. So much, in fact, that the process has nearly become its own domain of expertise, hence entire books dedicated to the subject. **The main challenge with this established domain is its ability to keep pace with changes in how modern applications and engineering are being performed:**

- Processes, methodologies, and recommendations are optimized towards obtaining the most comprehensive risk analysis and not necessarily cost-effective nor time-minimal uses in fast-moving engineering teams (CI/CD)
- Favoring of full/end-to-end system scope, which does not align with Agile, incremental, and other limited-scope engineering practices
- Total system ownership expectations that drastically increase modelling complexity when reviewing serverless application designs
- Desiring clearly delineated internal vs external boundaries that are significantly difficult to represent when using cloud service providers (CSPs) and other PaaS/IaaS services; the problem is compounded due to many CSPs having a publicly accessible management plane and other nuances causing a porous perimeter

The remainder of this article is going to look at how these challenges manifest when attempting to threat model a web application undergoing evolution from a classic design to a modern CSP-centric architecture.

“Note that a sufficiently robust design review process cannot be executed at CI/CD speed.”

Building Security In Maturity Model 9 (BSIMM9, AA1.2)
Threat Modeling a Classic Application

At the most basic level, threat modeling seeks to establish clear answers for the fundamental and necessary risk management questions:

- **What** are you building?
- **What** can go wrong?
- **What** can you do about it?
- **What** confirms the secure posture?

The questions seem simple, but can represent a lot of work to sufficiently answer. An application must be deconstructed into components and dependencies, and threat entry points (sometimes referred to as attack surface) must be enumerated, before it becomes clear what can potentially go wrong. Even the criteria for potential problems can vary based upon the focal point basis of the modeling:

- **Threat-centric** models consider adversary TTPs and motivations
- **System-centric** models consider the technical systems and functions
- **Asset-centric** models consider what needs to be protected

Multiple threat modeling methodologies leverage data flow diagrams (DFDs) to illustrate various application components, data flows between components, external entries into the data flows, trust boundaries, and attacker-worthwhile assets. DFDs focus on where data is coming from, where it is going, where it is stored, and how it is manipulated; the threat modeling exercise then looks at how attackers can affect those flows to influence/compromise data or gain access to an asset. DFDs can become fairly comprehensive, both in the expression of the application components and the illustration of threat avenues.

Process flow diagrams (PFDs) are a more recent invention, and resemble what a development or DevOps team would construct as part of normal design efforts. PFDs can cater to illustrating specific features and use cases (referred to as Application Threat Models or ATMs) or illustrate the operational topology of the application (referred to as Operational Threat Models or OTMs).
It is typical to maintain an OTM PFD to document operational and deployment items and leverage ATM PFDs for various application features; this makes ATM PFDs (along with the related Visual, Agile, and Simple Threat Modeling Methodology (VAST) approach) amenable to Agile practices and iterative application development.

Given the widespread coverage and varying focuses of threat modeling, this article is going to specifically look at design analysis threat modeling using a hybrid system- and asset-centric focus. This approach represents the most common type of activity an application development team would practically leverage during an architectural or design review of what they are building, allowing the technical team to focus on technical gaps and countermeasures agnostic of having intimate knowledge of the latest attacker TTPs. To showcase diagramming differences that occur in design analysis, both DFDs and PFDs are going to be utilized.

Introducing Our Analysis Target

Let’s look at a fictional but practical application scenario. A development team is responsible for a stateless, synchronous video encoding service implemented as a custom web application running on a typical web application framework/middleware server with a frontend web server (e.g. Nginx, Apache). The service simply receives uploaded raw video files, encodes them, and returns the results. The service does not leverage any authentication nor store any state (no databases, etc.), making it very simplistic overall. The security design analysis team proceed to construct a threat model DFD and PFD for their service (Figure 1).
The application seems simple, and the threat model diagrams mirror that simplicity. The team recognizes the web server as the main application entry point, and delineates the entry point trust boundary accordingly. They also realize the web server largely passes through the HTTP request to the application, thus the web application needs to be responsible for sanity checking incoming data before relying on it (thus mitigating various data injection attacks). The team considers denial of service, theft/abuse of services, system-level threats, and malicious internal personnel out of scope.

Do the team’s efforts seem reasonable? It appears straightforward, but threat modeling should encourage people to dig under the hood and ask deeper questions rather than be satisfied with a high-level picture. A seasoned security professional would know to ask:

- Is the web server frontend entirely stock? Are there any special configurations, plugins/modules, or changes done on that tier that should be reviewed or investigated?
- What is the communication path between the web server and application server, and is there any method to directly interact with the application server (e.g. listening on a different port)?
- What third party libraries, SDKs, and components are being used as part of the custom web application? It is entirely in-house code (except for the application server framework it builds upon)?

After receiving these inquiries, the design analysis team goes back and performs a deeper analysis. They decompose the web application into the main three parts the development team maintains: the web application framework handler (responsible for registering HTTP routes, receiving the HTTP request body content, sanity checking the incoming data, and passing the data along for actual processing), an in-house logic submodule library that does all the video encoding (non-HTTP aware, which is why it is wrapped in an HTTP handler), and a third-party video encoding SDK that the submodule relies upon. Upon investigating the third-party SDK, the team is surprised to learn the SDK actually leverages a third-party external SaaS service for video encoding (presumably to leverage faster GPU encoding hardware). The outbound communication from the SDK to the SaaS service could be subject to man-in-the-middle (MitM) or other network attacks. Fortunately, it appears the SDK is using TLS/HTTPS, but it is unknown whether the SDK is actually applying certificate validation, is subject to TLS downgrade attacks,
etc. Since the traffic is exiting the application back onto the external network, the team delineates another trust boundary in the DFD and marks that outbound connection as a risk.

The team updates their DFD and PFD to reflect their new findings (Figure 2). They also look into the communication between the web server and application server, and note that the application server binds to TCP port 5000 on localhost. Conceptually that means anything with localhost access can communicate directly with the application server, so they notate a localhost trust boundary in the DFD and document to always ensure the application server configuration specifies localhost network binding for the listening port, a mitigating security control to prevent the application server from accidentally listening on an accessible network port.

Things are looking much better, and the exercise to decompose and dig into the application has exposed hidden threats which now can be sufficiently managed or mitigated.
Clouds Roll In, and Bring Storms

Let’s take the same application development team and their video encoding web application from the previous scenario. The team’s service is experiencing great success, so they want to scale the application. As part of the effort, they also feel it is an appropriate time to move the application to a cloud security provider (CSP -- AWS in this case) and leverage the various AWS services. Unlike their prior legacy deployment which had IT managing the physical servers and networking the application was deployed upon, the move to a CSP leaves the application team responsible for full-stack management. Evolving from Dev into DevOps inherits additional infrastructure responsibilities and associated risks, which should be accounted for in the threat model accordingly.

The development team sat down and redesigned their application. The following major points were decided:

- The video encoding function in the legacy application (implemented in the submodule and SDK) are appropriate and thus will be retained; the team will not convert over to using AWS video encoding services.
- The HTTP frontend and web application handlers only exist to bridge HTTP requests to the video encoder submodule. As such, the team decides to use a serverless approach and implement the simple HTTP handlers as Lambda functions via AWS API Gateway.
- In order to work asynchronously, the team decides incoming raw videos will be staged into an S3 bucket. S3 will send file upload completion events to an SQS queue, and containers of video encoding workers (running via ECS) will poll the SQS queue for new files to download, encode, and reupload as results. Load scaling is achieved by launching more parallel worker containers.

But it’s worth mentioning that the analysis team’s success was significantly bolstered by the overall simplicity of the application, the application’s monolithic nature, and a consumable view of the entire computing ecosystem. These attributes are known to benefit threat modeling efforts; in particular, being feature and function complete in a system topology/ecosystem with little external entity reliance is the ‘best case’ situation for threat modeling.

Unfortunately, modern day computing is drastically heading in a different direction...
As part of the architectural effort the design analysis team creates a threat model DFD and PFD (Figure 3). They are sure to include known information and the concepts of trust boundaries from the previous application threat model. In this new topology, they consider colocation of all the components into a VPC as a designated trust zone, and delineate the VPC trust boundary in their DFD. Since everything is safely tucked into a VPC, and the configured method to enter the VPC (via the Lambdas) is through the API Gateway, the team rationalizes the only entry point into their ecosystem is through the API Gateway. The team also rationalizes they have nothing to do with the API Gateway, S3 service, and SQS service, so they depict those components as external entities in their DFD. The basic thinking is “it is not our code, it is not under our technical operation, we do not administer it, so it must not be ours.” This fits with one of the main tenet questions threat modeling seeks to answer: “What are we building?” The focus is scoped to what the application team is originating.

Since the topology has significantly expanded, the team also sits down and addresses scope changes. Moving to a multi-tenant CSP opens up new questions:
- Are threats from CSP personnel in scope? Typically, the CSP services are inherently trusted, which infers trust for the CSP personnel administrating or developing those services.¹
- Is multi-tenancy/co-tenancy an issue? Hypervisor, container, and sandbox escape vulnerabilities are rare but can happen; CPU hardware attacks and information leaks, like Spectre and Meltdown, can expose data cross tenant boundaries. Select CSPs offer single-tenancy and bare-metal hosting options if this is an area of concern.
- Are failures and bugs in CSP enforcement/operation of (mitigating) security controls an acceptable risk? Using CSP security mechanisms for your mitigating controls means you are taking on a cooperative partnership for your security posture.
- Are security controls provided by the CSP treated as external or internal entities? AWS security groups, IAM policies, and VPC containment become critical dependencies and first-class citizens in cloud-based architectures, yet would traditionally be labeled as external entities. Classically speaking, a design should not rely on external security controls to achieve its security posture.

All of these questions explore the notion of how security becomes a shared responsibility when using a CSP. Outside of extremely high security environments, the team takes a practical stance in addressing the risks related to this shared responsibility:

- CSP services are already trusted, so it is reasonable to trust CSP personnel too (and assume CSP will perform diligence against any internal personnel violation).
- Multi-tenancy using CSP standard hypervisor and container technologies is acceptable, with reasonable assumption the CSP is significantly motivated to address and prevent any threats at that level; therefore, multi-tenancy risks are considered out of scope.
- The team assumes any advertised CSP service will perform as documented and advertised, according to how it is configured. For example, IAM policy enforcement points will always correctly enforce a policy as stated (note: bad/incorrect policies are a separate issue!).
- Utilizing external security controls is the new norm, so the complications of shared security responsibility simply has to be accepted into the organization’s risk appetite as a cost of doing business in the cloud.

¹ Not including specialty security CSP services like CloudHSM and other security-vault technologies.
With CSP shared security responsibilities now stated, understood and accepted, the team is comfortable with the results shown in Figure 3 and feel AWS offered security controls offset the expansion in topology.

The question is: what did the design analysis team miss?

**Non-Obvious Entry Points and Trust Boundaries**

To the application developer, the first point of code execution is the Lambda handler function. An astute engineer may point out SQS events are triggered by AWS for S3 events, meaning the (trusted) AWS platform originates and manages all the incoming data to the worker container. That can be rationalized as no explicit trust boundary transitions or entry points once data enters the VPC via the Lambda handler front doors. Everything seems nicely contained behind security boundaries, and the only way to reach the custom application code is through trusted AWS services (API Gateway and SQS). All entry points seem accounted for.

The unfortunate reality is there are significantly more entry points present that are not being recognized. Many CSPs, and AWS in particular, have a publicly exposed management plane. APIs to publish SQS messages, invoke Lambda handlers, and upload S3 files are publicly accessible to the open Internet -- albeit gated by authenticated access and permissions. In fact, many AWS services are effectively publicly-accessible -- it is only through the use of VPC endpoints that a service appears to be contained within the VPC, when in reality the application pipeline is calling public APIs. The public entry point DFD for this design conceptually resembles what is shown in Figure 4.

“The threat modeling team should be critical when using external dependencies to mitigate threats.”

Swiderski & Snyder, Threat Modeling (Microsoft Press)
When looking at the scenario topology, here are some major entry points that are all accessible from the public Internet:

- **API Gateway** allows cross-account invokes; an attacker that can gain API gateway configuration access can effectively redirect requests to an attacker-controlled Lambda in another account and potentially establish a man-in-the-middle presence to all transactions.
- **Lambda functions** can be directly invoked by an attacker with Invoke IAM permission; this may break data trust assumptions by the Lambda handlers regarding the entities it assumes is invoking them.
- An attacker able to gain Lambda management permissions can update Lambda function code and push a new version, effectively gaining remote code execution access in the application transaction pipeline.
- **S3 buckets** have public endpoints, meaning attackers can directly list and access bucket files (subject to permissions); publicly accessible S3 bucket contents have become a notorious mechanism behind many recent data breaches.
- An attacker with appropriate SQS publish permission can inject arbitrary (malicious) events into the SQS event queue directly.

---

**Figure 4**
Depending upon EC2 instance networking configuration (e.g. public IP address assignment) and inbound security group configurations, the EC2 instance may publicly expose network service ports to an attacker; for the given topology, the output access by the worker container to the third-party SDK support service requires the instance to have a public IP address or subject the VPC to additional NAT setups.

An attacker able to gain assorted EC2 and ECS management permissions can drastically affect the operation of the worker container.

And lastly, an attacker capable of gaining IAM management permissions for IAM policies and roles, can leverage that access to achieve any of the prior listed items.

Expressing these entry points into our diagrams, the team is now dealing with the significantly more complicated diagram as shown in Figure 5. This is also where DFDs and PFDs start to diverge: the PFD retains simplicity by not capturing any of these dynamics in the diagram (rather, they are listed in
external/supporting exhibits), whereas the DFD visualizes the implications (by way of additional trust boundaries). The PFD simply arrives to the point where every entity in the diagram carries a threat or risk aspect that must be consulted elsewhere.

But let’s be practical and consider an obvious question: is this overkill? If we consider the concept of a classic network firewall, it seems obvious and thus pedantic that an attacker in the position capable of modifying the firewall configuration/ruleset can undermine the security control properties of the firewall. Isn’t attacker access to IAM management permissions effectively the same thing?

Perhaps, but the risk represents the cost of the entire attack tree (all the actions an attacker must take) to achieve the threat. For the classic firewall scenario, firewall configuration rulesets are often stored in internal repositories and take additional facilitation (manual or automated) to deploy to firewall nodes. In other words, there are additional internal barriers (i.e. defense in depth) that an attacker must overcome to cause a firewall ruleset change. IAM, on the other hand, is the conceptual equivalent to exposing your firewall administration service to the public Internet -- a single authentication token is all that sits between the attacker and making a ruleset change. The attack tree is significantly pruned, and any defense in depth capabilities are removed. Some may claim the attacker costs for the approximately equivalent threat have been lowered, thus affecting the risk consideration around how to deal with it.

Low-Risk Information Leaks Are Now Critical

As mentioned prior, the only thing standing in-between an attacker and a realized threat against the publicly-accessible CSP control plane is the possession of a properly permissioned authentication token. AWS assigns permissions to roles, and roles can be assigned to components (in this case, the Lambda handlers and the worker container). Under the hood, AWS is just providing these components with temporary authentication tokens. In our scenario design, the Lambda handlers are given a token with appropriate S3 access, and the worker container is given a token with appropriate S3 and SQS
access. Less obvious to engineers not intimately familiar with AWS is that these tokens are transferable and usable outside of the application environment. An information leak vulnerability that exposes a session token held by a Lambda handler can be leveraged externally to immediately access the S3 bucket, bypassing any safeguards or mitigating controls performed by the Lambda handler when accessing S3 assets. The same session token can also be abused, for example, to access requester-pays S3 content (the bill for the content will be charged to the account owning the session token used to authenticate). Overall this means traditional low-risk information leaks now carry a much higher severity if the leak can expose authenticated session tokens. In a certain sense, these tokens can reasonably be considered first-class assets needing protecting, warranting proper representation and analysis in the threat model.

All of Your Deputies are Confused

Because the data flow “pipeline” depicted in the DFD indicates multiple points of entry in the middle of the pipeline, the now porous application boundary is subject to a variable-trust dynamic whereby downstream components cannot necessarily determine if the upstream point of entry went through a trustable path. This is referred to as a “confused deputy” attack. For example, the payload provided to a Lambda invocation by API Gateway will carry a specific format containing AWS-originated values reasonably deemed trustable, and certain pass-through content from the public user that is known to be untrustable. If an attacker can directly invoke the same Lambda function, however, the contents of the payload are under arbitrary control of the attacker and everything is untrustable.

A more complicated example is found in the S3 event data flow to the worker container. When operating as designed, the Lambda handlers perform specific operations to the S3 bucket, which in turn generate S3 events delivered through SQS to the worker. The S3 event data is of a specific format, some of which is generated by the trusted S3 service and some which is pass-through from the S3 usage by the (trusted) Lambda handler. This may lead the downstream worker to trust the incoming event data, but in doing so it opens up risks due to confused deputy attacks: direct access to the S3 bucket, or direct publishing to the SQS queue can inject untrusted content that is
transparency forwarded through trusted entities deep into the application processing pipeline. Because there are untrusted entry points into the data flow, the trustability of the data payload and supporting metadata is questionable. The data will be trustable via certain flows, and untrustable via other flows, leaving the receiving application in a variable-trust situation when making trust considerations regarding the incoming data.

Other Deployment and Operational Factors

The presented design, and associated design analysis, were focused on the application operation once deployed. Not represented in the analysis are all the tightly-coupled processes, components, and CSP resources used as part of deployment and operation. For example, AWS Elastic Container Service (ECS) deploys containers from an origin repository (such as AWS ECR); attacker access to the container repository can affect the integrity of the code being deployed. CI/CD pipelines may stage binaries and built artifacts in additional S3 buckets as a means to transit application code into the CSP ecosystem; CSP autoscaling mechanisms essentially mandate a staged binary be readily accessible somewhere for instantaneous deployment. These storage locations are subject to similar access threats -- an attacker able to manipulate application binary artifacts in an S3 bucket or ECR repository will essentially gain remote code execution once the CSP system naturally deploys the code.

This is not a new dynamic per se, and once again seems aligned to how applications would classically be deployed. But just like the classic firewall example previously mentioned, the risk here is higher because the attack tree is significantly shorter: the production staged components are sitting in publicly accessible buckets and CSP infrastructure, gated by a single authentication token. Even worse, these infrastructure resources are typically where development to production transitions occur -- meaning access to these resources is often granted to developers, continuous deployment systems, and other developer environment integrations that often have relaxed security configurations and postures compared to production. It is common to see accidental leakage of developer access tokens (with permissions to production environment resources) in developer scripts, Jenkins configurations, and other technical materials uploaded to GitHub and storage locations.
and asking the right threat analysis questions to expose the answer. A design analysis team should be able to ask the right questions and discover risk about anything, without needing any special/prerequisite knowledge about what to ask (i.e. CSP operational particulars).

How to Evolve Design Analysis

Let’s be clear: the philosophies behind threat modeling and design analysis are not cloud-conflicting. Rather, it’s the historical ‘reduced to practice’ process prescriptions that tend to falter. Classic design analysis methodologies have evolved optimizations and shortcuts based upon the (assumed) foundation of discrete systems, networks, and databases in full possession of the owner. Shared security responsibility, external security controls, and publicly accessible CSP services/components significantly complicate an already complicated analysis process.

While it’s tempting to ignore these differences and generalize the circumstances back into classic non-cloud computing foundations, doing so will miss one of the largest emerging risks to data security: misconfigured or mismanaged CSP resources/assets. Be it exposed S3 buckets, open Elasticsearch clusters, or leaked AWS authentication credentials in GitHub, there are lots of new risks to consider, manage, and mitigate in these new topologies. Serverless platforms may reduce engineering complexity, but it does not fully eliminate the management risks.

In a perfect world, design analysis processes should naturally lead teams to recognize the risks that could lead to, for example, an exposed S3 bucket. That includes recognizing the trust boundary gated by IAM permissions, the bucket configuration itself, and the asset value of an upstream caller session token already having access permissions. This information can even be discovered without CSP-particular knowledge of how it all works: all it takes is selecting the right starting assumptions and asking the right threat analysis questions to expose the answer. A design analysis team should be able to ask the right questions and discover risk about anything, without needing any special/prerequisite knowledge about what to ask (i.e. CSP operational particulars).
It Starts with Ownership

When a CSP component is designated as an external entity to the application design analysis, this can lead to “out of scope” rationalizations that artificially dissuade any further investigation. That in turn can affect how the risks (or lack thereof) of a CSP component are documented and downstream responsibilities are planned to manage the component risk. In short, when a team looks at a CSP entity and declares “this is not ours, this is external,” the team is indirectly also saying “any related risks are not ours, they are external.” This can cause significant oversights.

A better approach is to consider all CSP entities as internal entities, which provides many advantages:

- In the “shared security responsibility” paradigm, it commits the application team to own final security responsibility in that partnership
- It prevents “out of scope” declaration shortcuts that could miss critical risk areas
- It can help ensure downstream processes (documentation, testing/validation, risk management planning, audit) to consider CSP component risk accordingly
- It ultimately sets the tone that the engineering team is accountable for the total security posture despite not being accountable for total operational implementation

It is less likely a team will accidentally forget to address S3 bucket security if appropriate management responsibility/consideration for the entire S3 service is had by the engineering team, rather than narrowly treating it like an internal file store hosted by an external entity.

Everything is Public, Until Proven Otherwise

It is very easy to overlook the publicly accessible nature of many CSP services used for internal application support. There are CSP services that specifically deploy into a contained network group
(such as an AWS VPC); but elsewhere there are services that operate as public endpoints. It is critical the design team understand the entry points supported by a particular CSP service or resource type, as there are often unenumerated entry points left unconsidered. Until enough experience is accumulated in dealing with particular CSP services, design analysis teams would be best served by assuming every CSP service exposes an undesired entry point of some sort -- leading the team to investigate what are the access control dependencies, mitigations, etc. For example, certain AWS services operate via public endpoints (e.g. DynamoDB, S3, SQS, Athena), some are hosted within a VPC (e.g. Aurora/RDS, Fargate, ElastiCache), and things like Lambda offer complicated hybrids with public endpoints that can operationally execute code within or outside your VPC. Access control mechanisms widely vary based on the nature of the service: sometimes you use IAM policies, sometimes you use security groups, etc.

Since CSP access control misconfigurations are causing many notable public data breaches these days (e.g. S3 buckets), taking the time to investigate the CSP resource to ensure proper understand of entry point capabilities and corresponding mitigating security controls is critical. This investigation is, fortunately, an infrequent exercise that produces highly reusable results. Organizations may benefit from creating one-time catalogs of risks and compensating control best practices for developers to consume -- after all, the risks of one particular S3 bucket are fairly typical of any S3 bucket use. BSIMM considers this a level 2 maturity item (Attack Model activities AM2.1 and AM2.2), and it is an essential necessity for standardization into further mature architectural analysis activities. There are also a few commercially-available threat modeling tools (such as ThreatModeler.com) that come pre-populated with similar known risk catalogs for specific CSP services.

Don’t Assume Trustable Data Origins

Due to the porous perimeter caused by publicly accessible CSP services, downstream application logic should not make any assumptions regarding format or trustability of incoming data payloads. A custom application worker assuming it is receiving an S3-generated JSON event blob over SQS
may be ill prepared to receive an arbitrary JSON (or non-JSON!) blob directly published to the SQS queue. Applications written to assume all incoming data as malformed and untrusted will be less prone to confused deputy attacks. Everything in the incoming data payload should be verified before use. For example, for an incoming AWS S3 event record:

- The application should validate the data actually represents an S3 event record, i.e. has the correct expected JSON schema, etc.
- The event principal ID, bucket name, owner identity, and bucket ARN can all be arbitrarily specified by an attacker directly publishing events -- thus the values should be sanity checked against whitelists, etc. before being considered for use
- The event name (S3 operation) and object key can also be used to indicate an object operation that didn’t actually occur; the application should not make assumptions regarding the key names generated by upstream components (e.g. the Lambda handlers), as an attacker can make direct S3 PUT requests with arbitrary key names in order to trigger injection attacks

Application logic that treats every incoming data payload as coming from a public, untrusted source will not just achieve good defense-in-depth practice, but will also create a resilient security posture less prone to cascading security failures and injection attacks if a CSP resource (access control) becomes misconfigured.

## Data Is the Infrastructure

The last thing design analysis teams should consider is the role critical pieces of data, such as access control policies and deployment configurations, play in defining the infrastructure.

Simply put: your security group policy (data), for all intents and purposes, is your firewall (infrastructure) in the cloud. Your deployment specifications (CloudFormation, Terraform) and autoscaling configurations will directly dictate your infrastructure topology. Your IAM policies explicitly define your application and infrastructure perimeter. As the world continues to embrace serverless
computing, the metadata definition of an application operation (typically bundled with serverless configurations/deployments) will be the closest representation to “infrastructure” you have. Recognizing the importance of configuration data will help drive two key actions:

- Infrastructure configuration data can now be modeled as an asset with appropriate safeguards and threat considerations
- Changes to infrastructure configuration data effectively change the infrastructure, which may trigger/warrant a security re-analysis of the architecture

**Putting It All Together**

The purpose of these suggested methodology additions is to help teams jettison historical shortcuts in threat analyses and naturally bring the right attention into CSP resource threat areas that appear to be plaguing organizations. Unsecured cloud infrastructure has emerged as a designated threat category, and attackers are specifically capitalizing on the new-found opportunity. Cybersecurity risk management is already hard enough for application teams, and the introduction of paradigm-changing cloud and serverless infrastructures throw a monkey wrench into the classic prescribed approaches to address infrastructure and application security postures.

Make no mistake: the use of CSP services, serverless, and IaaS increase the complexity of security risk management despite CSPs offering reduced complexity of engineering management. Threat modeling and design analyses are going to inevitably require an increased investment and adjustment due to shared security responsibilities -- but the goal is finding the methodology that provides a path to a strong security posture with least additional cost and investment into niche technical expertise of CSP-specific underpinnings. A solid design analysis approach, in combination with CSP service threat catalogs and strong consideration to atypical application needs (porous infrastructure, untrustable data, etc.) will provide the best generalized, reusable approach multiple teams can leverage as they assess their risk on their journey into the cloud.
About the Author

Jeff Forristal

Jeff is an expert cyber security technology leader/advisor known for identifying/creating new technical security service offerings, developing industry-first security product features, and driving research into new industry areas. He has operated as technical contributor, technical director, and executive/CTO for small security product startups to very large Fortune 500 finance and technology organizations.

He has a wide background in software + firmware + hardware, technology architecture, operations, and applied security strategy.

He is an accomplished thought leader and writer, having written multiple features and cover-story articles for Network Computing and Secure Enterprise magazines; he is also a contributing author to multiple books. Under the pseudonym “Rain Forest Puppy,” he has been recognized as an industry expert in web application security and was responsible for noted industry landmarks including the first documented discovery of SQL injection, the first responsible security disclosure policy and the first intelligent web application scanner. In addition, he has presented his research in many forums, from established security events like RSA, BlackHat and CanSecWest to smaller regional conferences around the world.