PReVeNT REAL CLOUD ATTACKS WITH Terraform

SANS CLOUD SECURITY

HashiCorp Terraform

OpenTofu

HashiCorp Terraform and its open-source forks, such as OpenTofu, are infrastructure-as-code tools. By defining your infrastructure in HashiCorp Language (HCL), Terraform will shape your environment to match your code. While it can help your organization consistently enforce security and compliance, it also has security considerations of its own. This poster covers all of the concepts you need to work in the Terraform ecosystem and how it can be used to mitigate attacks against infrastructure in the Big 3 cloud providers: Amazon Web Services (AWS), Azure, and Google Cloud.

In **SEC510: Cloud Security Controls and Mitigations™ (sans.org/sec510)**, we use thousands of lines of Terraform code Terraform is used to build a complex, multi-cloud lab environment comprising hundreds of resources across AWS, Azure, and GCP.

As one student put it: "If I was given the exercise objectives and told to create a single environment for just two of them, I would be hard pressed to do so. The fact that every single exercise can be run across all three Tier 1 cloud service providers is astounding."

Terraform was key to developing SEC510's lab environment. After the authors determined what they wanted to build and had some practice with the syntax, the code practically wrote itself. This poster distills key lessons, best practices, and common pitfalls uncovered during the development of this real-world environment.

Originally created to support the SEC510 course, this resource also serves as a valuable companion across the broader SANS Cloud Security curriculum (sans.org/cloud), where Terraform is widely used.

Terraform Core Concepts

Most Common Terraform Block Types

`terraform` and `provider`

These are used to list and configure the *Terraform Providers* used by this project. Providers are plugins that interact with cloud providers, on-premises services, and more. Providers created by HashiCorp and others are published on the Terraform Registry. Once defined, these providers can be installed with the `terraform init` command.

Terraform is often described as "cloud agnostic," meaning it can be used to seamlessly interoperate with the different cloud providers. Many security professionals erroneously believe that their organization can produce a single set of Terraform code to configure them all securely. They might assume that you can create a storage bucket in each of the Big 3 cloud providers like so:

IDEAL

```
resource "storage_bucket" "assets" {
  name          = "assets"
  target_clouds = ["aws", "azure", "gcp"]
}
```

In reality, you would need different, provider-specific resources for each cloud:

REALITY

```
resource "aws_s3_bucket" "assets" {
  bucket = "assets"
}

resource "azurerm_storage_account" "assets" {
  name = "assets"
}

resource "google_storage_bucket" = "assets" {
  name = "assets"
}
```

Each cloud's provider is completely different. In fact, Azure has at least three unique providers: `azurerm` for resource management, `azuread` for configuring Entra ID, and `azapi` for Azure API calls. Google Cloud has at least two: `google` and `google-beta`. Visit the following URL to learn more about how Terraform is Cloud Devout, not Cloud Agnostic: **sec510.com/agnostic**.

The following blocks will prepare Terraform for initialization so that it can work with AWS, Azure, and Google Cloud:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "X.Y.Z"
    }

    azurerm = {
      source  = "hashicorp/azurerm"
      version = "X.Y.Z"
    }

    google = {
      source  = "hashicorp/google"
      version = "X.Y.Z"
    }
  }

  # Store the Terraform state file in Amazon S3
  backend "s3" {
    profile = "default"
    key     = "tfstate"
  }

  # Other options include storing the state in an
Azure Storage container using "azurerm" or Google
Cloud Storage using "gcs"
}

provider "aws" {
  profile = "default"
}

provider "azurerm" {
  features {}
}

provider "google" {
  credentials = "/path/to/service-account-key-file.
json"
}
```

There are at least three security questions to ask about this code:

1. **Do these providers enable supply-chain attacks?** These providers may contain vulnerabilities or malware.
2. **Am I storing my Terraform state file securely?** These files contain many details about your deployment and may contain secrets. Terraform is storing this state file in a bucket within the Amazon Simple Storage Service (S3). If this bucket is public, these secrets would be exposed to attackers.
3. **How is Terraform authenticating to the cloud?** Terraform needs Identity and Access Management (IAM) permissions to perform deployments. The Google Cloud provider is using a plaintext key file to authenticate its identity. It would be better to use temporary credentials, such as those from Workload Identity Federation.

`storage_bucket`

These define what Terraform should create. In general, a Terraform resource maps to a single item in the cloud. For example, a storage bucket and a storage object (file) are two Terraform resources.

These blocks contain the resource's desired configuration. These are usually provided as *arguments* which are key-value pairs, but they can also be blocks that contain arguments. All resource blocks have the following format:

```
resource "resource_type" "local_name" {
  argument = value

  block {
    argument = value
    ...
  }

  ...
}
```

A resource's arguments can be referenced elsewhere in the codebase (`resource_type.local_name.argument`). When created, these resources will generate *attributes* that can also be referenced. For example, you can reference the Amazon Resource Name (ARN) of a newly created bucket using `aws_s3_bucket.local_name.arn`. The documentation for every resource provides an exhaustive *Argument Reference* and *Attribute Reference* section.

The `resource_type` must exactly match the name of a resource in the Terraform provider. The `local_name` is arbitrary. Sometimes, they match the name of a resource:

```
resource "google_storage_bucket" "photos" {
  name = "photos"
}
```

This is equivalent to:

```
resource "google_storage_bucket" "whatever_label_i_want" {
  name = "photos"
}
```

Use the local naming scheme that is easiest for you. Just like naming variables in a normal programming language, there is no perfect solution. While you can use `camelCase`, `PascalCase`, or `kebab-case`, because Terraform uses `snake_case` consistently, we recommend doing the same. Because Terraform is *declarative*, the order in which the resources are declared usually does not matter.

No two blocks can use the same resource type and local name. For example, this is not allowed:

```
resource "google_storage_bucket" "bucket" {
  name = "bucket1"
}

resource "google_storage_bucket" "bucket" {
  name = "bucket2"
}
```

This is allowed, however, because the resources with the local name `bucket` have different resource types:

```
resource "aws_s3_bucket" "bucket" {
  bucket = "bucket"
}

resource "google_storage_bucket" "bucket" {
  name = "bucket"
}
```

`variable`

These define a value that can be supplied during the deployment, which Terraform can then reference. They are often used to eliminate hardcoded values and secrets. The values can be provided interactively during the deployment, the `.tfvars` file, or via environment variables starting with `TF_VAR_`. If a variable's value is not provided, the default defined in the code will be used. Otherwise, Terraform will throw an error.

```
variable "allowed_cidr_block" {
  description = "The CIDR block from which to allow access to the
resources. By default, they will be publicly accessible."
  type        = string
  default     = "0.0.0.0/0"
}
```

`output`

These indicate which values generated by Terraform should be exported. These values can be read by scripts outside of Terraform. **These values can be sensitive.** While those with the `sensitive` argument set to `true` will be masked by default, they can still be read from the state file just like everything else created by Terraform.

```
output "azure_client_id" {
  description = "The Client ID of the Azure Service Principal
created by Terraform."
  value       = azuread_service_principal.my_principal.client_id
}

output "azure_client_secret" {
  sensitive   = true
  description = "The Client secret of the Azure Service Principal
created by Terraform."
  value       = azuread_service_principal_password.my_principal.
value
}
```

File Structure

While many Terraform users will put all of their code in a single `main.tf` file, this can get messy quickly. Code should be broken down into logically grouped files. Terraform will effectively concatenate every `.tf` file in the deployment directory. The names are completely up to the user.

`variable` and `output` blocks are usually defined in files named `variables.tf` and `outputs.tf`, respectively. This makes it easier for the user to see exactly what to provide and what they will receive during a deployment. In the SEC510 course, we place all of the `required_providers` and `provider` blocks for a given cloud provider in `providers.tf`. For everything else, here are two approaches that you can use when appropriate:

- `<service_name>.tf`—All code for a specific cloud service. Examples include `s3.tf`, `iam.tf`, and `vpc.tf`.
- `<product_name>.tf`—If multiple cloud services are required to create a single product, it might make sense to put them in the same file. For example, in the SEC510 course, we have an AWS Lambda fronted by an Amazon API Gateway, so we put the resources for both services in a file called `functions.tf`.

There are a very small number of reserved Terraform file names. This includes `_override.tf` and `override.tf`. Putting resource blocks in either of these files will override the provided arguments for resources defined in the other Terraform files. This is great for patching configuration without having to modify the main code itself.

Associating Resources

To put an object in a bucket, we associate these resources by providing the bucket name in the object's configuration:

BAD

```
resource "aws_s3_bucket" "document_
storage" {
  bucket = "documents"
}

resource "aws_s3_object" "example" {
  bucket  = "documents"
  key     = "example.txt"
  content = "Here are the file contents."
}
```

Associating resources with a hardcoded name is not best practice. If we change the bucket name, the object will still point to the old bucket name and fail to be created. Additionally, Terraform will not be aware that these resources are related. This can cause a *race condition* where the deployment may fail trying to create the object before the bucket is created. Both issues can be resolved by referencing the `id` attribute of the bucket resource in the `bucket` argument for the object resource:

GOOD

```
resource "aws_s3_bucket" "document_
storage" {
  bucket = "documents"
}

resource "aws_s3_object" "example" {
  bucket  = aws_s3_bucket.document_
storage.id
  ...
}
```

For another example, we start by creating a network in the Virtual Private Cloud (VPC) service:

```
resource "aws_vpc" "main" {
  cidr_block = "10.42.0.0/16"
}
```

We then create a security group, a stateful firewall capability in AWS, within the VPC with a rule allowing HTTPS from the world:

```
resource "aws_security_group" "https" {
  vpc_id      = aws_vpc.main.id
}

resource "aws_security_group_rule" "https_
from_the_world" {
  security_group_id = aws_security_group.
https.id
  type              = "ingress"
  from_port         = 443
  to_port           = 443
  protocol          = "tcp"
  cidr_blocks       = ["0.0.0.0/0"]
}
```

Finally, we associate the security group with a virtual machine instance in the Elastic Compute Cloud (EC2) service, exposing the VM to the world:

```
resource "aws_instance" "app_server" {
  ...

  vpc_security_group_ids = [aws_security_
group.https.id]
}
```

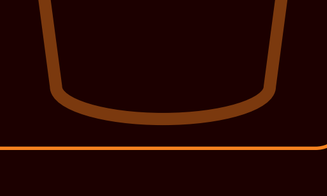
`depends_on`

Terraform will automatically create a dependency graph based on the attribute references. This will usually prevent race conditions. However, there are exceptions. For example, Amazon S3 Object Lock is a security control that prevents file object versions from being deleted or modified. It requires Object Versioning to be enabled. These are managed with two Terraform resources that do not reference each other. If Terraform creates the Object Lock resource first, it will result in the following error:

"InvalidBucketState: Versioning must be 'Enabled' on the bucket to apply an Object Lock configuration"

We can add an explicit dependency using the `depends_on` clause. The following will ensure that the Object Versioning resource is created before the Object Lock:

```
resource "aws_s3_bucket_versioning" "assets" {
  bucket = aws_s3_bucket.assets.id

  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_object_lock_configuration" "assets" {
  depends_on          = [aws_s3_bucket_versioning.assets]
  bucket              = aws_s3_bucket.assets.bucket
  object_lock_enabled = "Enabled"
  ...
}
```



Terraform Techniques, Tips, and Tricks for Security Professionals

Detect and Resolve Configuration Drift

One of Terraform's core responsibilities is to generate a *plan*. For the initial deployment, Terraform will create all of the resources defined as code. It will generate a *state file* that contains all of the exported attributes for every resource. If the user changes the code and deploys, Terraform will generate a plan determining which resources need to be created, updated, or destroyed in order for the cloud to match the code. When plans work as planned, Terraform is *idempotent*, meaning that performing multiple deployments will only have an effect if changes are necessary.

Various Terraform commands use plans. `terraform plan` generates and prints the plan. `terraform apply` generates the plan and performs the deployment based on the plan. `terraform plan -destroy` and `terraform destroy` perform the same operations respectively, but for destroying the infrastructure.

One of Terraform's goals is to prevent *configuration drift*, where the desired configuration does not meet reality. This improves consistency, helps teams move faster, and eliminates *ClickOps* (i.e., manual changes made by engineers in the cloud user interface). To help prevent drift, at the beginning of the planning process, Terraform performs a *refresh*. This attempts to match the state file to the actual cloud environment, updating the state file when differences are detected. Refreshes can be run without generating a plan by using the `terraform refresh` command.

Unfortunately, Terraform's refreshing capability is quite limited. It often requires manual intervention to fix the state file, especially when Terraform times out while creating a resource. Most importantly, **Terraform will not detect resources that it did not create**. If an attacker creates a malicious resource in the target's cloud account, such as a virtual machine running a cryptocurrency miner, Terraform alone will not detect it. Here are some tips for addressing this issue.

1. **Monitor your environment for unmanaged resources.** This can be automated using various tools, such as Cloud Security Posture Management (CSPM) tools that can generate an inventory.
2. **Update Terraform state to track externally created resources.** If you have a legitimate resource that was created outside of Terraform, you can import it into state using `terraform import`. This is a manual process that requires you to supply the resource's Terraform name (`resource_type.local_name`) and the cloud's identifier for that resource, such as an ARN, an Azure resource path, or Google Cloud resource URL. This can also be accomplished in the code itself using an `import` block. Similarly, you can explicitly change the address of a resource using the `terraform state mv` command or `moved` block.

Do Not Manage Secrets with Terraform

Everything generated by Terraform must be stored in the state file in order for the plan to generate properly. This includes secrets. For example, this code will create an AWS IAM user and credentials for that user:

BAD

```
resource "aws_iam_user" "my_user" {
  name = "my_user"
}

resource "aws_iam_access_key" "my_user" {
  user = aws_iam_user.my_user.name
}
```

If other code needs to reference this user's Access Key ID and Secret Access Key, it can use `aws_iam_access_key.my_user.id` and `aws_iam_access_key.my_user.secret` respectively. **This is only possible because these secrets are written to the state file.** It will look like the following:

```
{
  "module": "module.aws",
  "mode": "managed",
  "type": "aws_iam_access_key",
  "name": "my_user",
  "provider": "module.aws.provider.aws",
  "instances": [{
      "attributes": {
        "id": "AKIA...",
        "secret": "Cx9...",
        ...
      },
      ...
  }]
}
```

If you need to use secrets, they should be generated out-of-band and stored in secrets managers, such as HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, or Google Cloud Secret Manager. Note that **by default, Terraform will store these values in the state file even if it's initially pulling them from one of these services**. This can be mitigated by using the `ephemeral` block type, which functions like a resource but has a unique lifecycle and does not get stored in state.

If you absolutely need to generate an Access Key ID and Secret Access Key in Terraform, the AWS provider supports the `pgp_key` argument. This takes a Base64 PGP key or a username for Keybase.io and uses it to encrypt the Secret Access Key. If this argument is used, the `secret` attribute will be replaced with `encrypted_secret`. However, this just pushes the problem one level down as you now need to protect the PGP key.

Destroy Insecure Default Resources

Terraform is great for creating and managing resources. It's not nearly as good at managing resources that already exist. New cloud accounts come with default resources, many of which are insecure. We can import them into Terraform, but this can be complicated. In many cases, it's better to delete these default resources altogether.

To do this, we can use the `null_resource`. This acts like any other resource, but it does not directly create a resource in the cloud. Instead, we can use it to orchestrate operations using a `provisioner`. For example, the `local-exec` provisioner runs a script on the machine performing the Terraform deployment. There is also a `remote-exec` provisioner, which runs a script on an external system. While Terraform recommends against using `null_resource`, sometimes there is no viable alternative.

The following example deletes the default firewall rules and VPC from a Google Cloud project:

```
resource "null_resource" "delete_dangerous_
default_vpc" {
  provisioner "local-exec" {
    command = "(gcloud compute firewall-rules
delete default-allow-icmp -q || true) && (gcloud
compute firewall-rules delete default-allow-rdp -q
|| true) && (gcloud compute firewall-rules delete
default-allow-ssh -q || true) && (gcloud compute
firewall-rules delete default-allow-internal -q
&& gcloud compute networks delete default -q ||
true)"
  }

  triggers = {
    always_run = "${timestamp()}"
  }
}
```

It runs multiple commands in succession to delete these resources. They are chained together with the `&&` (and) operator. Each command uses the `||` (or) operator to ensure that, even if the commands fail, Terraform will consider this "resource" to have been created successfully. This is necessary as the second time this provisioner is run, the commands will fail due to the firewall rules and VPC no longer existing, which is fine and does not indicate that the deployment failed.

The `triggers` object ensures that this command will be run on every deployment. By default, the provisioner would only run when the `null_resource` is being created. After our initial deployment, if we recreated these firewall rules or VPC, our next deployment would not remove them. So, we set `always_run` to the current timestamp. Because this will be different on each deployment, this command will be run for every deployment.

Beware `jsonencode`

`jsonencode` is a function that translates a Terraform object into JavaScript Object Notation (JSON). It is quite useful, especially for applying IAM policies in AWS. Still, it has been used incorrectly in at least one case causing a security incident.

In September 2023, researchers from Datadog, Wiz, and others disclosed the misuse of `jsonencode` in a popular tutorial for implementing workload federation between AWS and GitHub Actions. The flawed policy allowed an attacker to create a GitHub Action, assume a role into any AWS account with this implementation and at least interact with private AWS CodeCommit resources. This issue was also found in an open-source repository from the United Kingdom's Government Digital Service (GDS) agency. Here is the vulnerably code:

BAD

```
assume_role_policy = jsonencode({
  Statement = [{
    ...,
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "token.actions.githubusercontent.
com:sub": ["repo:GitHubOrg/GitHubRepo:ref:refs/
head/GitHubBranch"]
      },
      "StringEquals" : {
        "token.actions.githubusercontent.
com:aud": "sts.amazonaws.com"
      }
    }
  }]
})
```

This Terraform code is defining the trust policy for a role assumed by GitHub Actions. This IAM statement's condition contains two different StringEquals fields. JSON does not allow an object to have two different values for the same field. As a result, the jsonencode function will ignore the first instance of the field and only serialize the second instance. This eliminates the condition that checks whether the request is coming from the appropriate GitHub repository and branch. The correct way to implement this condition is to have a single `StringEquals` object in the statement that validates each value:

GOOD

```
assume_role_policy = jsonencode({
  Statement = [{
    ...,
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "token.actions.githubusercontent.
com:sub": ["repo:GitHubOrg/GitHubRepo:ref:refs/
head/GitHubBranch"],
        "token.actions.githubusercontent.
com:aud": "sts.amazonaws.com"
      }
    }
  }]
})
```

Users have requested a Terraform for misconfiguration like this. Follow the GitHub issue here: **sec510.com/jsonencode-issue**

Several fixes have since been implemented. GDS collaborated with Datadog to patch its repository. The tutorial creator worked with Scott Piper from Wiz to resolve a similar issue. Scott also encouraged AWS to continue working on its end to mitigate this issue. AWS then launched a new version of the trusted entity setup wizard that prompts the user to specify the GitHub organization, repository, and branch to trust. AWS notified customers who did not contain a condition evaluating the `sub` claim on August 18, 2023. As of November 1, 2023, this policy is enforced on all these policies. Learn more about this case study and its mitigations here: **sec510.com/jsonencode-mitigation**

Apply Critical Security Controls

SEC510™ teaches students how to prevent real attacks using controls that matter, many implemented using Terraform. Here are some of the author's favorites:

Prevent Service Secrets from Being Accessed Outside of the Private Network

AWS Secrets Manager

```
data "aws_iam_policy_document" "secretsmanager" {
  statement {
    actions = ["secretsmanager:GetSecretValue"]
    resources = ["*"]
    effect = "Deny"

    condition {
      test     = "StringNotEquals"
      variable = "aws:SourceVpc"
      values   = [aws_vpc.app.id]
    }
  }
}

# Allow resources in the VPC to connect to the
Secrets Manager over a private IP.
resource "aws_vpc_endpoint" "secretsmanager" {
  vpc_id              = aws_vpc.app.id
  service_name        = "com.amazonaws.<Region>.
secretsmanager"
  vpc_endpoint_type   = "Interface"
  subnet_ids          = [aws_subnet.private1.id,
aws_subnet.private2.id]
  security_group_ids  = [aws_security_group.
secretsmanager.id]

  # Automatically migrate to the private endpoint
without code changes.
  private_dns_enabled = true
}
```

Azure Key Vault

```
resource "azurerm_key_vault" "sec510" {
  ...

  network_acls {
    default_action = "Deny"
    bypass         = "AzureServices" # Required for
app gateway to read the TLS certificate.
  }
}

resource "azurerm_private_endpoint" "vault" {
  ...

  private_service_connection {
    name                           = "sec510-vault-
endpoint"
    private_connection_resource_id = azurerm_key_
vault.sec510.id
    subresource_names              = ["vault"]
    is_manual_connection           = false
  }
}

# Automatically migrate to the private endpoint
without code changes.
resource "azurerm_private_dns_zone" "sec510" {
  name                = "sec510${var.unique_string_
azure}.vault.azure.net"
  resource_group_name = azurerm_resource_group.
sec510.name
}

resource "azurerm_private_dns_a_record" "sec510" {
  name                = "@"
  zone_name           = azurerm_private_dns_zone.
sec510.name
  resource_group_name = azurerm_resource_group.
sec510.name
  ttl                 = 300
  records             = [azurerm_private_endpoint.
vault.private_service_connection[0].private_ip_
address]
}
```

Securely Perform Integration from Azure to AWS with Workload Identity

```
locals { azure_identity_audience = "api://sec510" }

data "aws_iam_policy_document" "azure_function_
assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRoleWithWebIdentity"]

    principals {
      type = "Federated"
      identifiers = [aws_iam_openid_connect_
provider.azure_tenant.arn]
    }

    condition {
      test     = "StringEquals"
      variable = "sts.windows.net/${var.azure_
tenant_id}/:aud"
      values = [
        local.azure_identity_audience
      ]
    }

    condition {
      test     = "StringEquals"
      variable = "sts.windows.net/${var.azure_
tenant_id}/:sub"
      values = [var.azure_identity_principal_id]
    }
  }
}

resource "aws_iam_openid_connect_provider" "azure_
tenant" {
  url = "https://sts.windows.net/${var.azure_
tenant_id}/"

  client_id_list = [local.azure_function_identity_
audience]

  # Thumbprint for sts.windows.net

  thumbprint_list =
["626d44e704d1ceabe3bf0d53397464ac8080142c"]
}
```

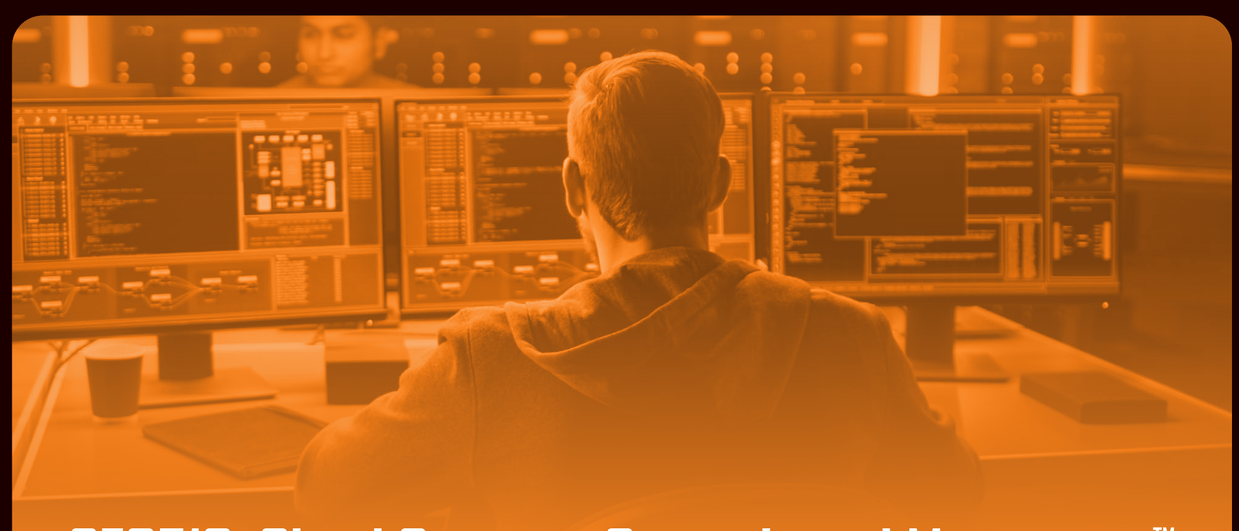
Replace the Extremely Dangerous Default Google Cloud Function Service Account

```
resource "google_cloudfunctions_function"
"document_upload" {
  ...

  # <Project ID>@appspot.gserviceaccount.com has
the Editor role.
  # This allows full read and write access of
everything except for IAM!
  # Replace the service account to use the
principal of least privilege.
  service_account_email = google_service_account.
functions.email
}

resource "google_project_iam_custom_role" "storage_
creator" {
  role_id     = "storageCreator"
  permissions = ["storage.objects.create"]
}

resource "google_storage_bucket_iam_member"
"storage_creator_documents_functions" {
  bucket = google_storage_bucket.documents.name
  role   = google_project_iam_custom_role.storage_
creator.id
  member = "serviceAccount:${google_service_
account.functions.email}"
}
```

Scan Your Terraform Code

Terraform is only as effective as your code. There are several tools available to scan your code for vulnerable configurations. These tools include:

Checkov:	www.checkov.io
DeepSource:	deepsource.com
EasyInfra:	github.com/SeisoLLC/easy_infra
Terrascan:	github.com/tenable/terrascan
Trivy:	trivy.dev



SEC510: Cloud Security Controls and Mitigations™

GIAC Public Cloud Security (GPCS)

Protecting multicloud environments takes more than default settings and compliance checklists. SEC510 focuses on attack-driven cloud security controls that work across the Big Three Cloud Service Providers (CSPs) to reduce misconfigurations and vulnerabilities. The course highlights the limitations of CSP defaults, industry frameworks, and benchmarks, while equipping professionals with practical tools to lower risk and protect critical assets. Application flaws are inevitable, but with the right cloud controls, they're far less likely to lead to a breach.

sans.org/sec510

sans.org/cloud-security

@SANSCloudSec

youtube.com/SANSCloudSecurity

linkedin.com/showcase/sanscloudsec

SANS CLOUD SECURITY

This poster was created by SANS Instructor Brandon Evans with support from SANS Cloud Security Faculty.

CS_SEC510_Terraform_0525