AWS Ephemeral bastion hosts

Introduction

In every organization there are always situations where an administrator, developer or DBA requires access to resources, either for maintenance, troubleshooting or to run some diagnosis. Company security policies normally dictate that these resources remain private and that they are to be accessed from controlled environments with tight access controls.

To reduce the attack surface, proxies are deployed in private environments with security measures in place that allow:

- Single point of access to private networks.
- Centralized access control policies.
- Software hardening of the proxies meant to grant access to the private networks.

Bastion hosts are machines (virtual or bare metal) that are deployed centrally or spread through different environments with the purpose of behaving as doors to otherwise inaccessible resources.

In order to get access to private resources, a user requires access to these hosts. Authentication normally takes the form of either SSH keys or some sort of authentication system that recognizes the user and grants him access to the host. Given an organization tends to have different environments where software is deployed (which in the case of big companies this can be multiplied by the number of different projects they run, each one with their own set of private environments), the number of hosts deployed (and the amount of maintenance and the cost that that implies) could quickly complicate the task of maintaining this infrastructure. Also, distributing the required SSH keys to the hosts so users can connect to them can easily become a nightmare as these keys need to be rotated to avoid compromising the infrastructure after a theft.

Managed Services to the rescue

In recent years, AWS has announced different services to help organizations to deal with the problem of accessing private resources without the need to maintain either SSH keys or bastion hosts all together.

First it was [Systems Manager Session Manager](), which gave users access to instances without the need for SSH keys or open ports in security groups. The only requirement was to have the Session Manager agent installed on the instance and connected to the service so that users could connect to it using their AWS IAM credentials.

Then EC2 Instance Connect was announced, which gave users the ability to connect using SSH to instances with short lived keys generated on the fly. This still required opening ports to grant access to the instance but removed the burden of maintaining the keys separately.

Finally, at the end of 2020, AWS announced AWS CloudShell, an interactive terminal directly on a web browser with user sessions capable of storing up to 1GB of data persistent between session creations. AWS has promised to improve the service in future iterations, including the possibility to deploy the backend instance inside a VPC which is not possible at the moment.

Another approach

Given all the solutions provided by AWS described above, there is still space for improvements. AWS Systems Manager Session Manager and EC2 Instance Connect still require a permanent instance running to connect to. They are still the solution to situations where users need to access resources directly in a host without the need for a jump box but this is not normally the case in the current era of containers (Fargate doesn't even allow SSH connections even though there are ways to manually install it) or where a DBA/Developer needs a terminal to run some SQL queries on a database from. For cases where a bastion host is still required, these systems still require a machine running 24x7. AWS CloudShell has the potential to be the place to go once some extra features are added like the possibility to connect to a VPC.

The following solution makes use of AWS Systems Manager Session Manager to connect to a host. But in this case the host is created on the fly and destroyed once the job is done. This provides benefits like:

- Reduces the cost of maintaining EC2 instances running 24x7.
- Removes the need to apply patches to the hosts or to replace them with more up to date images. New hosts are created from the latest version of an AMI.
- Removes the need to keep individual users inside the host or share the same one for every connected user.
- Every session starts clean. Files or applications installed by users during a session are removed at the end.
- Possibility to create bootstrap actions. These would automatically install the required software for a session. Another option would be to create different AMIs for each use case but it is more difficult to maintain.

Apart from these, there are improvements to the model to increase the security posture for those companies that require it.

- Commands could be logged and sent to S3 to keep track of user actions.
- A snapshot of the instance could be created at the end of the session for forensic analysis in case necessary.
- Session files could be sent to S3 at the end of the session for each user so following sessions could start with the files kept in a specific folder.

Architecture

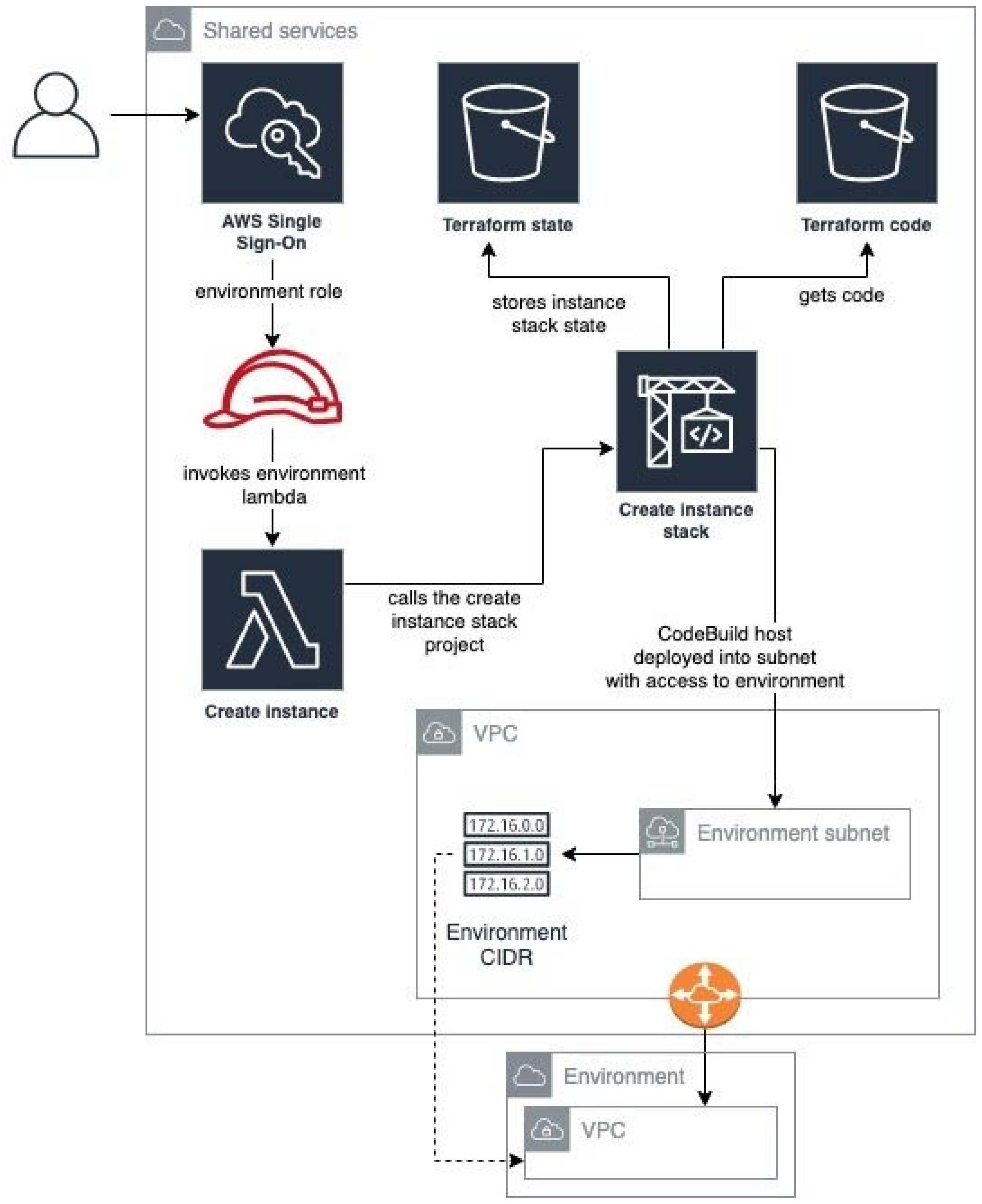
The proposed solution follows the AWS convention in terms of separation of concerns. It requires a central AWS account (called in this case Shared Services) bastion hosts are deployed on. They connect to each environment, each one deployed in a different AWS account, through a VPC peering connection. To make sure a bastion host deployed to connect to one environment cannot in fact connect to a different one through the VPC peering connection, they are deployed in different subnets, each one with a route table associated that grants access only to the right CIDR range for the target environment. Users are handled using AWS Single Sign On which in turn can be configured to connect to an existing AD or used alone.

Not all companies follow this approach. Some don't have a different AWS account per environment. Some may prefer to deploy a bastion host per environment directly inside the environment itself. Even though this solution doesn't support that, it can be adapted with relative ease. The bulk of the work is done by services like Lambda, Systems Manager Session Manager, DynamoDB, IAM, CodeBuild, CloudWatch Events or S3. The network topology can be adapted to the requirements.

One thing to keep in mind is that the deployment is done through Terraform, configured to use S3 as a backend. As explained before, this can be adapted. But in this case it would mean more work to translate it to something like CloudFormation.

Let's describe each of the elements of this architecture in the chronological order they are used.

Bastion host creation

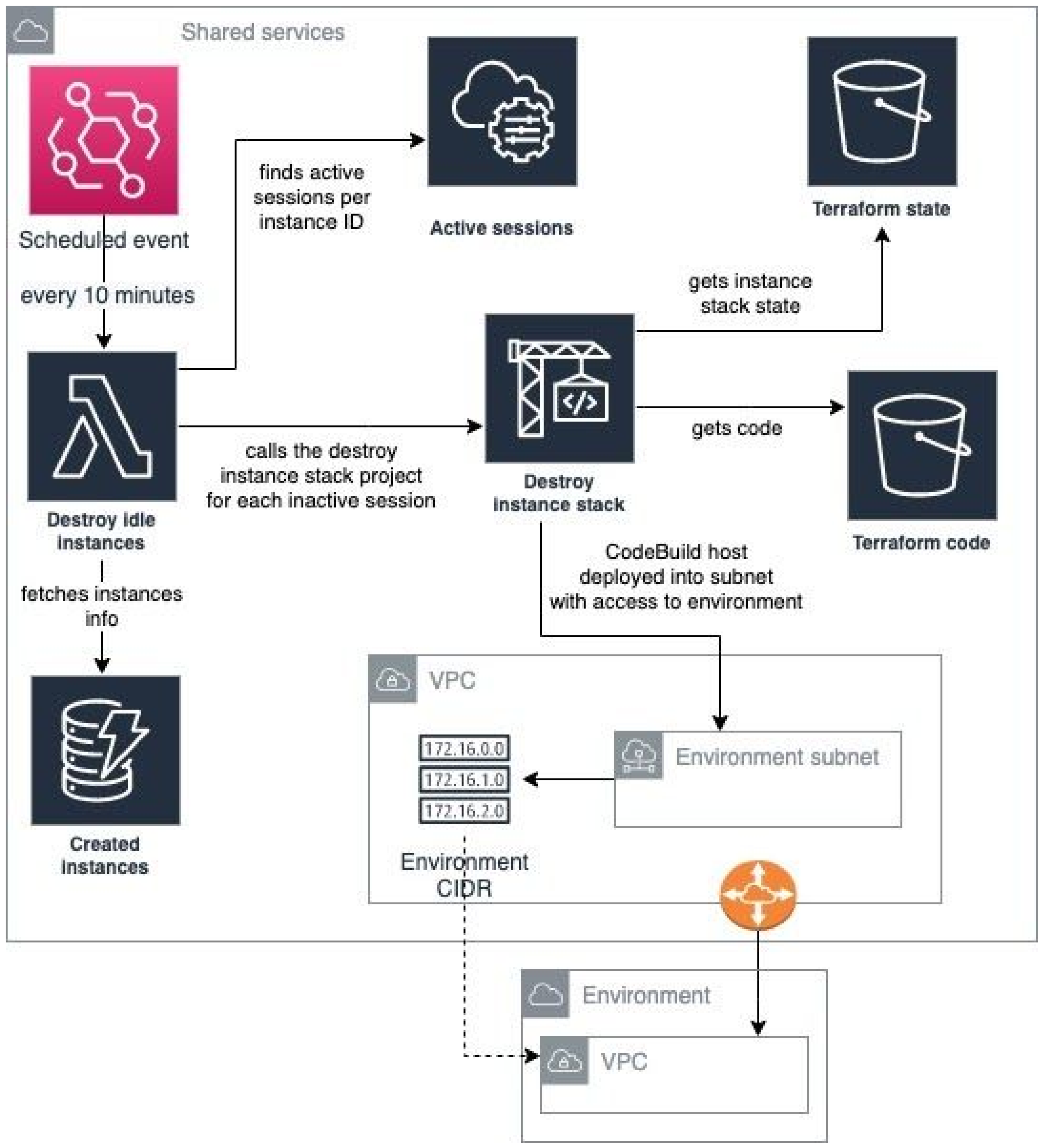


- Users that require access to an ephemeral host are created and controlled from AWS Single Sign On (SSO). This service provides the required access control that allows administrators to grant access to the environments from a central place.
- Each user is assigned one or many permission sets in SSO. These permission sets contain a policy that grants Lambda invocation and Systems Manager Session Manager execute access.

- There is a ‘Create Instance’ Lambda per environment. The permission sets described above allow each the execution of a single Lambda. That’s how control over which environment each user can create a host on is handled. The ‘Create Instance’ Lambda function is responsible for 2 things:
    - It generates a unique UUID. More on this in a moment.
    - It calls a CodeBuild project that executes a Terraform script.
- The ‘Create Instance Stack’ CodeBuild project is passed in some variables from Lambda like the environment it has to deploy the host on and the UUID. The UUID is used as an identifier for Terraform. In order for the stack to be destroyed after the bastion host is no longer in use, we need a way to map the created instance and its stack to a Terraform tfstate file. For the EC2 instance, we have its instance ID. For the tfstate file the way the architecture handles it is by creating a Terraform workspace where the workspace name is the UUID from Lambda. That way, when the stack is destroyed, another CodeBuild job can simply change to the workspace for that stack and execute a ‘terraform destroy’ command.
- There is a DynamoDB table whose purpose is to store all these references. The item creation is done by the Terraform script itself after the instance has been created. The information stored in each item include:
    - The environment the host was deployed for.
    - The instance ID.
    - The UUID to identify the Terraform tfstate.
    - The source version of the code that created the stack. This is stored so changes to the Terraform code deployed while hosts are running don’t block the destruction of the stack.
- Once the instance is created and connected to Systems Manager Session Manager, users can connect to it using the AWS CLI. As the command to connect to the instance requires its ID, the Terraform script adds a tag to the host with the UUID generated by the Lambda as a value, which is returned in the response from the function after a successful invocation. Using the EC2 describe-instances command, a user can easily find the instance ID by filtering by this tag.

  There is a script as part of the source code that takes care of all the orchestration of calling the Lambda function, getting the resulting UUID and then using it to connect to the instance using Session Manager once it’s available.

Bastion hosts destruction

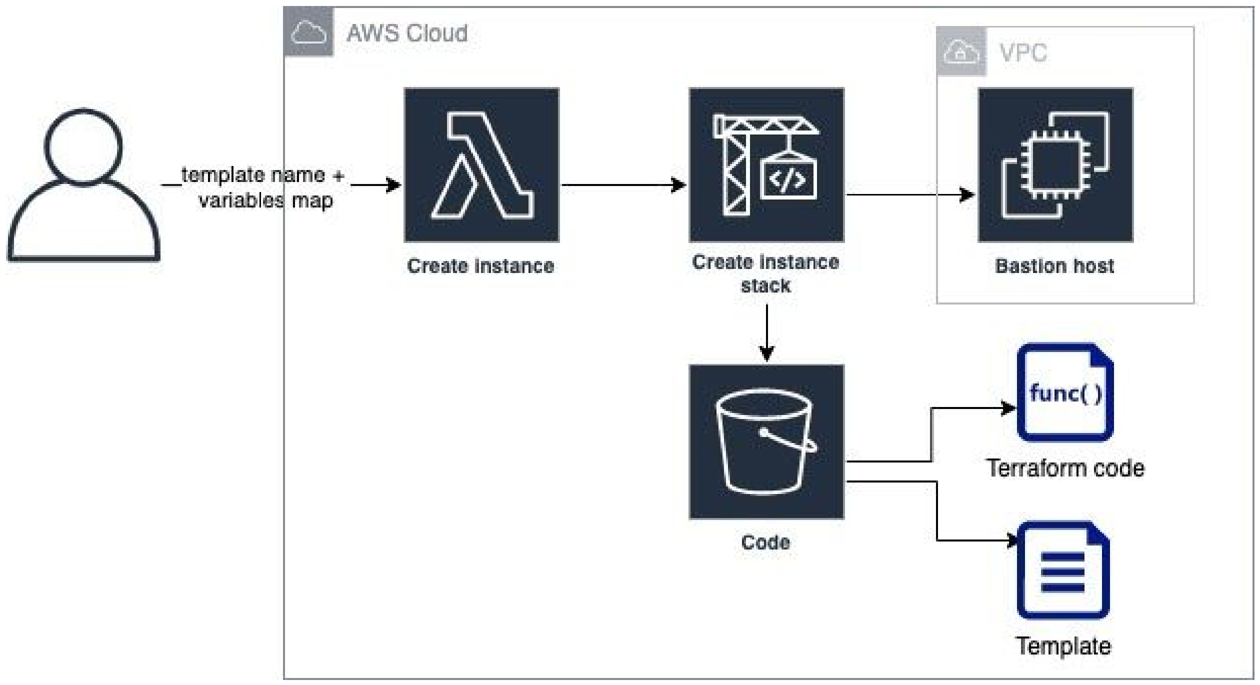


- Every 10 minutes (configurable) a CloudWatch Event is triggered to invoke the Lambda function ‘Destroy Stack’.
- This Lambda function finds all the currently active Systems Manager Session Manager active sessions and gets all information about created stacks from DynamoDB. With that then it finds which instances have no running sessions and triggers the CodeBuild project ‘Destroy instance stack’ for each one of them.
- The CodeBuild project takes the UUID and the environment passed to its execution, gets the Terraform code and executes the ‘terraform destroy’ command after changing to the workspace for the instance ID identified by the UUID.

Bootstrap actions

As described when listing the benefits this architecture provides, there is the option to include bootstrap actions, not explained above. This is currently done using EC2 user data to pass a bootstrap script to be executed after the instance has been created. It is often the case that these scripts require some kind of external input in the form of variables. To overcome this problem, the architecture allows the creation of templates with the bootstrap code where variables are injected to the Terraform script from the ‘Create Instance Stack’ CodeBuild project, which takes them from the ‘Create Instance’ Lambda which in turn gets them from a map passed by the user with the values for these variables and the name of the template to use.

The following diagram describes the process.



There are currently 2 templates available with the option to create more if needed.

- *database_tunnel*: it installs the linux application *socat* in the host. Once the bastion host is created, it creates a socat service that forwards a port in the host to a URL (in this case a database connection URL) and starts it. This way, a user can locally configure a JDBC connection to localhost and the configured port to get a proxied connection to the database through the bastion host and run SQL queries from his RDBMS tool of choice.
- *install_git_terraform*: it installs Git and Terraform in the host and configures a Git user automatically. It can be modified to include a list of repositories to clone as part of bootstrapping the host to increase development speed.

Conclusion

The AWS ecosystem provides many feature-rich services to simplify the lives of developers and system administrators. The hurdle that’s always been keeping systems secured but at the same accessible for administrative tasks nowadays has been greatly simplified by cloud providers. They have done all the heavy lifting required to give us secured environments on which to deploy our infrastructure and have given us the access control services that let us manage our private resources in a simpler way. But as with everything, there is always room for improvement and some use cases require some extra work to adapt to the requirements.

This solution builds on the fantastic tools already available to give response to those cases not yet covered, waiting for a future out of the box service that will simplify it even more.