AWS - Mandatory Tests

Introduction

Every AWS pen-test requires that we complete AWS Mandatory Test Cases and for us to fill out a form indicating that we completed them. Even for projects where the mandatory test cases do not apply, we still need to fill out the survey.

The survey can be found here: https://mandatory-test-cases-form.aka.amazon.com/ (Midway protected, internal AWS network access only)

In the "Related Ticket" field, fill out the section with your SIM ticket number, e.g. AVP-1234

If a test case does not apply, do not fill anything out in the section and do not tick the "Does this test case apply to scope the scope" box.

1. Verify that the API action validates the caller's IAM permission

Verify the rules of implicit and explicit denies are followed. That means that if a user/role doesn't have an allow for an action, they're implicitly denied. If they have a statement in their policies that denies an action (explicit deny), then they cannot do that action no matter what the other policies say. Suggested test cases for each API call:

- Explicit allow rule (action should be allowed)
- Wildcard allow with explicit deny (action should not be allowed)
- No allow or deny rule (action should not be allowed)

2. Verify that the caller has valid permissions to pass a role when providing a role to the service

Verify that users/roles/groups have the iam:PassRole permission in order to pass a role to some service. If a service lets you pass a role to some other service (the other service performs an action on the requestor's behalf under the passed role), like EC2 or Lambda, then iam:PassRole is required. Suggested test cases for each API call that passes a role:

- Explicit allow PassRole:RoleName rule (should work)
- Wildcard allow PassRole:* with explicit deny PassRole:RoleName (should not work)
- No PassRole permission specified (should not work)

3. Verify that the role passed by the caller belongs to the same account as the caller

For any service where you can specify a role as part of a command, the role needs to come from the same account as the caller. If you can pass a role from other accounts, that's a huge red flag for danger. This is one of the most common "confused-deputy" issues that always concern AppSec. Suggested test cases for each API call that passes a role:

- Explicit allow PassRole: rule in user's IAM policy, or a user with administrative access, attempts to pass a role from another account (should not work)

4. Verify that all constraints in the resource policy for a resource are validated for every action against that resource

If resources can have policies attached to them, like with S3 buckets or KMS keys, then the rules of IAM resource policies should apply. This generally means that an applicable deny statement in a resource policy is an explicit deny, an allow statement in a resource policy grants access even if a user/role /group policy (in the same account) doesn't have an allow. But, an explicit deny in a user/role/group policy overrides an explicit resource allow. And, if there's no resource policy, access should be dictated by ownership of the resource. Suggested test cases for each action that can be set in the resource policy:

- Explicit allow in resource policy, empty user policy (should allow access)
- Wildcard allow and explicit deny in resource policy, empty user policy (should prevent access)
- Empty resource policy, empty user policy (should deny access)
- Empty resource policy, explicit allow in user policy (should allow access)
- Explicit allow in resource policy, explicit deny in user policy (should deny access)

5. Verify that the caller has valid permissions to call the service to perform actions on a resource

Make sure that the resource part of an IAM user/role/group policy is correctly checked (should already be supported natively by IAM, but you never know). The service description for a customer facing service can be found here: https://naps.amazon.com/service_description_files (also see 'Tips' section below). This file should be updated prior to launch . Ensure enforcement matches what is recorded. Suggested test cases for each action that a user can perform on a resource:

- Wildcard allow for any resource (should work)
- Explicit allow rule for the action on a different resource (should not work)
- Explicit allow rule for the action on the target resource (should work)
- Wildcard allow with explicit deny for the target resource (should not work)

6. Verify that the service expects a role from the caller with a policy that adheres to principle of least privilege

Some services allow users to set up a role to grant access to their accounts to the service, usually with one or no clicks. Make sure that role isn't granting unnecessary access, give it a smell test.

7. Verify that adding a principal aspen policy to a resource does not escalate the privileges of the caller

For resource policies, they need to have a principal that the statements apply to. Also, if you have a service named abc, then you shouldn't be able to add a resource policy that grants access to xyz and then get that access. When granting access to a resource via AssumeRole, make sure that the requested IAM policy does not give permissions that the access policy of the role does not allow. Suggested test cases for each resource which can use a resource policy:

- Resource policy allows access to the resource it is attached to (should work)
- Resource policy allows access to a different resource (should not work)
- Resource policy allows access to a resource belonging to a different service (e.g. s3 bucket) in this account (should not work)

8. Verify that the service validate resource permissions/ownership when performing actions on the resource

Make sure that someone can't pass an ARN for a resource that belongs to someone else's account during an API call. The service should be checking for those shenanigans. This is another "confused-deputy"-type issue. Suggested test cases for each call that acts on a resource:

- Pass a resource this IAM user has access to (should work)
- Pass a resource a different IAM user in this account has access to (should not work)
- Pass a resource from another account with a policy allowing this account and user (should work)
- Pass a resource that belongs to a different account (should not work)

9. Verify that when a service interacts with the customer's S3 bucket, it checks for bucket ownership

Going along with the previous test case, given accounts A and B, which both give a service access to the S3 objects in their accounts via a service role, then account A shouldn't be able to send a request that specifies an S3 object in account B. This is another "confused-deputy"-type issue. Suggested test cases for each call affecting an S3 bucket:

- Specify an S3 bucket this user has access to (should work)
- Specify an S3 bucket in the same account this user does not have access to (should not work)
- Specify an S3 bucket in another account that this user should have access to (should work)
- Specify an S3 bucket in another account that this user should not have access to (should not work)
- If the service performs prolonged actions (e.g. writes to the bucket every hour) then see if the bucket can be deleted from an authorized account and created with the same name in another account, and still receive data (bucket sniping)

10. Verify that actions in a service are using the right source IP information to evaluate IAM policy

Policies let you add conditions based on the requestor's IP address (and others globally and per service). Check that this condition is correctly checked. The reason this test case is here is because it is possible the API request goes through multiple hosts before AuthN/AuthZ checks are done. If not set up correctly, the check would be looking at one of these hosts addresses instead of the requestor's. If authentication might be forwarded, verify that the X-Forwarded-For header cannot be used to confuse the endpoint. One approach is to add conditions to the IAM Allow policy, allowing several minutes between each to allow the policy to propagate. For example, if you are testing from 192.168.1.1:

- IpAddress 192.168.1.1 (should work)
- IpAddress 127.0.0.1 (should not work)
- Add X-Forwarded-For: 127.0.0.1 header to the request (should not work) NotIpAddress 127.0.0.1 (should work)
- NotIpAddress 192.168.1.1 (should not work)

Note: Appsec recommends verifying this from an EC2 instance, as source IPs from VPN never seem to be right. When testing on-site, the IP you get from googling "what's my IP" will probably be different than what the AWS API will see due to some proxy weirdness. Use the EC2 console; there's a thing that will let you know which IP the AWS API will see you as when on-site.

11. Verify that the service handles protocol specification appropriately

Ensure that the services front end load balancer and back end service handle the HTTP protocol specification identically. A malicious user should not be able to abuse differentials in the specification to illicit unintended behavior.

Suggested test cases for abusing specification differentials:

- Smuggling an HTTP request through differential handling of Content Length and Transfer Encoding

Tips

- Ensure that you clear out your previous policies while testing.
- You can use the EC2 console to find out your internal IP as seen by AWS. Go to EC2 > Security Groups > Create Security Group. Click Add Rule, and select "My IP" for the source. It will fill in a small range in the nearby dialog box. (If this still doesn't work try launching an ec2 instance.) Alternatively, you can make a CLI call and view the SourceIp for it in CloudTrail.
- IAM policies take time to propagate at times. Sometimes up to 5 minutes. **BUT**, if you create a new set of access keys after editing a user's IAM policy, the new policy should be completely functional immediately with these new keys.
- For Test Case #11, refer to the following blog posts describing the attack:
  - https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
  - https://portswigger.net/research/http-desync-attacks-what-happened-next
  - Or our internal wiki: Request Smuggling with HTTP Desynchronization
- For Test Case #11, services using the following version of JLBRelay should not be vulnerable, but dynamic testing is still recommended-- https://w .amazon.com/bin/view/JLBRelay/Releases/October_2019_Dingo/
- For Test Case #5, not all resources support resource-level permissions. The service definition for a service can be found here: https://naps. amazon.com/service_description_files . This specification propagates to the IAM visual editor and public documentation, so if a resource is specified in the 'RequiredResources' or 'OptionalResources' section for an API resource-level permissions should be supported. The public facing documentation for how optional vs required resources work can be found in the second paragraph. here: https://docs.aws.amazon.com/IAM/latest /UserGuide/list_amazons3.html#amazons3-actions-as-permissions .
- For Test Case #10, If you need to figure out what IP amazon sees your IP as inside of corpnet, and have access to a server in corpnet, you can use tcpdump to track your host's IP via ping: sudo tcpdump icmp -n (credit to Maxfield Chen)

Deprecated Guidance on AWS Mandatory Test Cases

This section is maintained for historical purposes. While it may be useful for understanding the mandatory test cases, it should be considered deprecated.

1. Action-Level Permissions

Verify that the API action validates the caller's IAM permission

> *Zak: Check that every externally accessible API in scope properly denies access when there is no explicit Allow or an explicit Deny in the IAM policy.*
>
> *Subu: Yes, and that if there is an explicit deny, the IAM user is not allowed to call that API action.*

Test Steps:

Test 1
- Add an IAM policy to awstest1 explicitly allowing access to the service (EC2, RDS etc.).
- Ensure that the awstest1 user can invoke the service API.
- Repeat this test for all the APIs.

Test 2

- Add an IAM policy to awstest1 allowing wildcard access to everything.

- Add an IAM policy to awstest1 explicitly denying access to the service (EC2, RDS etc.).

- Ensure that the awstest1 user can no longer invoke the service API.

- Repeat this test for all the APIs.

Test 3
- Remove the previous IAM policy. Add an IAM policy to awstest1 explicitly denying access to the service.
- Ensure that the awstest1 user can no longer invoke the service API.
- Repeat this test for all the APIs.

Test 4
- Remove the previous IAM policy.
- Ensure that the awstest1 user can no longer invoke the service API, even though there is not a policy explicitly allowing or denying these actions.
- Repeat this test for all the APIs.

2. Role Passing

Verify that the caller has valid permissions to pass a role when providing a role to the service

> *Zak: Check all APIs that pass in a role properly deny access when there is no explicit Allow (*) or an explicit Deny for the role in the IAM policy*
>
> *Subu: Not quite. In order to pass a role to an AWS Service, the called must have the iam:PassRole permission. If they don't have this permission, they are not allowed to pass the role. The permission can be scoped down to specific roles an IAM user can pass. Essentially, passing a role allows an IAM user to implicitly give permissions to an AWS service, which could lead to privilege escalation. So, this should only be allowed if the IAM user has the authority to do so.*

Test Steps:

Check if the API has an argument that allows an IAM role to be passed. If there is no such API, this test is N/A.

- Add an IAM policy granting the user administrative access but explicitly denying iam:Passrole.
- Ensure that awstest1 cannot invoke the API with the role argument and attempt to pass a role even against owned resources.
- Ensure that all the APIs that pass a role are tested.

3. Role Ownership

Verify that the role passed by the caller belongs to the same account as the caller

> *Zak: Check all APIs that pass in role, can only pass in roles that have the same account ID.*
>
> *Subu: Yes. I would reword it as all API actions that accept a role from the user check that the role being passed by the user belongs to the account of the calling user. Cross account passrole is a terrible design decision and should not be done.*

Test Steps:

Check if the API has an argument that allows an IAM role to be passed. If there is no such API, this test is N/A.

- Create a role in awstest2 and record its name/ARN. This is Role2.
- Ensure an awstest1 IAM user has IAM:Passrole permissions allowed
- Invoke the API with the role argument as Role2 as awstest1. This should NOT work.
- Ensure that all the APIs that pass a role are tested.

4. Resource-Based Permissions

Verify that all constraints in the resource policy for a resource are validated for every action against that resource

> *Zak: Check that APIs properly denies access when there is an explicit Deny in the resource policy. What should we do about resources that cannot have resource policies attached to them?*
>
> *Subu: Its not just DENY policies but also ALLOW. Resource policies make access control decisions based on a multitude of attributes like request source IP for S3 buckets. This test case is expected to cover all of them. When there is no resource policy, resource access should be governed by ownership. This test case will not apply for services that don't support resource policies.*
>
> *Tom: … is specifically referring to resource-based permissions… as defined at: http://docs.aws.amazon.com/IAM/latest/UserGuide /reference_aws-services-that-work-with-iam.html … most pen tests will answer #4 as N/A as most services do not support resource-based permissions*
>
> *Subu: Yes, you are correct.*

Notes

Note that this test step is specifically about resource-based permissions, i.e. policies attached directly to service resources (not to IAM users/roles) regarding who can do what with them. Definitions and interactions with resource-based permissions vary based on the service, with S3 bucket policies being a most well-used example.

Test Steps:

- Verify that the API performs an action on a specific resource. If it does not, this test is N/A for that API.

- Verify that the service supports resource-based policies. If it does not, this test is N/A. Here is a link that mentions which services support resource based policies: http://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-services-that-work-with-iam.html (This isn't always completely up to date so do your own research as well

- Sometimes the service supports resource-based policies and has an applied policy but you **cannot** edit it. In such cases do the tests below but by manipulating your user's policies so that they are more restrictive than the applied resource-based policy and checking that the request is allowed/denied accordingly. Do whatever tests make sense in this scenario.

Test 1: Resource Allow, User Empty (ALLOW)

- Identify a resource in awstest1's account. Record their ARN.
- Create a resource policy granting the user access to just that resource.
- Create a user with an empty resource policy.
- Verify that the user can access only that resource.
- Repeat this test for all the APIs that interact with that resource.

Test 2: Resource and User Empty (DENY)

- Identify a resource, record the ARN.
- Set an empty policy on both the resource and the user
- Verify that the user cannot access this resource.
- Repeat for all relevant APIs

Test 3: Resource Allow w/ Explicit Deny, User Empty (DENY)

- Identify a resource, record the ARN.
- Set a wildcard allow for all users and an explicit deny for awstest1 in the resource policy.
- Insure that the user has an empty policy set.
- Verify that the user cannot access this resource.
- Repeat for all relevant APIs

Test 4: Empty Resource Policy, User Policy has Explict Allow (ALLOW)

- Identify a resource, record the ARN.
- Set an empty resource policy
- Set an explicit allow for the given resource in the user policy
- Verify that the user can access only resource.
- Repeat for all relevant APIs

Test 5: Explicit Allow in Resource, Explicit Deny in User (DENY)

- Identify a resource, record the ARN.
- Set an explicit allow for your chosen user in the resource policy
- Set an explicit deny for the given resource in the user policy
- Verify that the user cannot access the resource.
- Repeat for all relevant APIs

Test 6

- Attach a resource constraint that a valid user can meet, to the resource policy created in Test 1.
- Here is a list of all the constraints that one can keep: http://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_condition-keys.html#AvailableKeys
- Verify that users who meet the set constraint can perform an action on the resource.
- Repeat this test for all the APIs that interact with that resource.

Test 7

- Verify that users who do not meet the set constraint cannot perform an action on the resource.
- Repeat this test for all the APIs that interact with that resource.

5. Resource-Level Permissions

Don't skip this test case

There is a new (as of September 28, 2018) campaign to have ALL new services support resource-level permissions (among other requirements) before they launch. All existing services must meet these (internal Amazon link) requirements by February 15, 2019.

Verify that the caller has valid permissions to call the service to perform actions on a resource

> *Zak: Check that APIs properly denies access when the "Resource" field is not explicitly set to Allow (*) or an explicit deny for the resource in the IAM policy. What should we do about services that do not implement Resource based IAM policies?*
>
> *Subu: Yes. Resource condition in IAM policies should be supported on most services cause IAM natively supports it. When this is not the case, document it and ignore this test case.*
>
> *Tom: … is referring to resource-level permissions, as defined at: http://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-services-that-work-with-iam.html … most should test #5 as most services support resource-level permissions.*
>
> *Subu: Yes, you are correct.*

Notes

Note that this test step is specifically about resource-level permissions, i.e. using the "Resource" section of an IAM policy to define which service resources (identified by ARN) a IAM policy statement applies to.

- Verify that the service supports resource-level policies. If it does not, this test is N/A. Here is a link that mentions which services support resource based policies: http://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-services-that-work-with-iam.html

Test Steps:

Test 1

- Identify the API that performs an action on a specific resource.
- Identify a resource in awstest1's account. Record their ARN.
- Configure an IAM policy allowing the user access to only that resource.
- Ensure that the user can perform actions only on that resource.
- Repeat this test for every resource that the API interacts with.

Test 2
- Configure an IAM policy denying the user access to only that resource.
- Ensure that the user cannot perform any actions on that resource.
- Repeat this test for every resource that the API interacts with.

6. Least Privilege Roles

Verify that the service expects a role from the caller with a policy that adheres to principle of least privilege

> *Zak: We are unsure about the suitability of this test case. We can check that roles created by the service team or expected for the service to not have administrative access. However, this seems to be primarily a business decision to be made by the service team.*

> *Subu: The role created by the service (1/0 click role creation) or the one the user passes in should be limited to what the service needs. This is usually documented in the threat model. I'm asking for a sniff test here to ensure that the role is limited in scope based on the pen-test engineers understanding of how the service works. If it doesn't smell right, talk to the service team and AppSec and help us understand what feels wrong and if there is a reasonable justification.*

Test Steps:

- Study the threat model and other documentation to identify the name of the role the service account uses.
- Obtain access to the service account's console via Isengard (ask the service team for RO access to their production account).
- Once logged in as the service user, navigate to the IAM Role page and study the permissions of the role the service uses.
- Ensure that the permissions are similar to what the threat model says the permissions should be.
- If you do not get Isengard access to the service account, ask the service team to extract the permissions the service account has and verify that against the Threat Model.

7. Resource-Based Permission Evaluation Order

Verify that adding a principal aspen policy to a resource does not escalate the privileges of the caller

> *Zak: Check that an Allow resource policy with a "Principal" field is set and an IAM policy with an explicit Deny is set for the service call, results in a deny.*
>
> *Subu: Yes and one more thing. Sometimes when services expect a resource policy (like S3 bucket policy), the calling user can provide an IAM policy with ALLOW IAM:* permissions in an attempt to confuse the policy evaluation engine to apply the principal policy to a resource thereby getting the principal (caller) access to iam:*, which is obviously bad. This should be tested.*

Notes

I believe that #7 also refers to resource-based permissions. See #4 for more details. -Tom

If your service does not support this, this test is N/A. -Arvind

Sometimes the service supports resource-based policies and has an applied policy but you **cannot** edit it. In such cases do the tests below but by manipulating the aspen policy so that it is more restrictive than the existing resource-based policy and checking that the request is allowed/denied accordingly. Do whatever tests make sense in this scenario.

Test Steps:

Test 1
- Add a resource-based policy allowing a user access to a resource
- Add a resource-level/IAM policy denying the action
- Verify that the user does not have access to perform an action on the resource
- Repeat this test for all the APIs that interact with that resource

Test 2
- Change the resource-based policy to deny access to a user
- Change the resource-level/IAM policy, granting the user iam.* permissions
- Verify that the user does not have access to perform an action on the resource
- Repeat this test for all the APIs that interact with that resource

8. Direct Resource Access

Verify that the service validate resource permissions/ownership when performing actions on the resource

> *Zak: Check that APIs which can pass in ARNs properly denies access to resources that they do not have permission for.*
>
> *Subu: Yep.*

Test Steps:

- Identify a resource in awstest1's account. Record its ARN.
- As a user in awstest2's account, invoke an API that performs an action against the resource in awstest1's account.
- Ensure that this access is denied.
- Repeat this for all the possible resources awstest1 can own across all the APIs in scope.

This is very similar to any normal horizontal authorization bypass test that you do on any project.

9. S3 Bucket Ownership

Verify that when a service interacts with the customer's S3 bucket, it checks for bucket ownership

> *Zak: This test case seems very similar to 8), and how should globally accessible S3 buckets bet taken into account?*
>
> *Subu: Think about a bucket sniping situation. In most cases, this would be covered by 8, however, in some cases the service maintains access to the bucket to interact with in in the future (unprompted by the user). For example, I set up cloudfront access logging to an S3 bucket and when I do, cloudfront validates that I own the bucket and sets up access to write to the bucket. I forget this and delete the bucket. A bucket sniper, snipes this bucket and marks it for unrestricted write to the cloudfront service account. Now attacker gets my cloudfront access logs and I am none the wiser. This needs to be tested.*

Test Steps:

- Identify an S3 bucket in awstest1's account. Record its ARN.
- Make this bucket world-readable and world-writeable.
- As a user in awstest2's account, invoke an API that performs an action that uses the bucket owned by awstest1.
- Ensure, as awstest1, that the bucket state is still the same after the action was performed by awstest2.
- Repeat this for every action made while interacting with an S3 bucket.

10. SourceIP Condition

Verify that actions in a service are using the right source IP information to evaluate IAM policy

> *Zak: Check that APIs use the service caller's IP address when using the "Condition" field and "aws:SourceIP" key.*
>
> *Subu: Yep*

Test Steps:

Note: If you are on the Amazon internal network, the best way to find the IP address that AWS APIs see when you make a request is to go to the EC2 console, click on "Security Groups", click on the "Inbound" tab, click "Edit", then "Add Rule", then under "Source", select "My IP". The IP address should then appear next to the "Source" column.

Note: This makes sense to test in production. Just be careful while testing, from a penetration test perspective. Also, make sure to clean your credentials out of your public IP address, once you are done testing – you do not want credentials lying around all over the place.

Note: If this is an internal IP address, ask for Beta/Gamma access and study the logs to see where your request is originating from. If you cannot get access, ask the service team to monitor your requests and tell you what IP you are coming from.

Test 1
- Setup awscli on a publicly accessible IP address (like situx) with your credentials
- Set an IAM policy granting access to the entire service.
- Add a condition mentioning the source IP that the API can be invoked from to this policy.
- Invoke the API from some other IP address.
- Ensure that this request fails.
- Repeat this test for every API in scope.

Test 2
- Invoke the API from the source IP address (situx) that was set in the policy.
- Ensure that this request succeeds.
- Repeat this test for every API in scope.