Pivoting in Amazon Clouds

Introduction

Mission critical applications are being deployed to the Amazon cloud and most information security experts have no clue about what needs to be inspected to make sure they are secure.

As we'll learn from this research, classic security testing is not enough, knowledge about Amazon's EC2 instance life-cycle, user-data, IAM roles, and other Amazon cloud services are required when testing and exploiting Amazon cloud architectures.

Tools and PoC code will be released as part of this research. The tools, written in Python using the boto library, provide the following features:

- Enumerate access to AWS services for current IAM role
- Use poorly configured IAM role to create new AWS user
- Extract current AWS credentials from meta-data, .boto.cfg, environment variables, etc.
- Clone DB to access information stored in snapshot
- Inject raw Celery task for pickle attack

Instance meta-data

All EC2 instances have meta-data, such as the used AMI, kernel and region. This meta-data is made available to the instance through a web server (only accessible to that particular instance) which lives at http://169.254.169.254/ . Amazon's meta-data documentation better explains all the details about the instance meta-data and how to access it.

From the information security perspective the important information available in the meta-data is:

- Local IP Address
- User-data
- Instance profile: AWS API credentials as explained in *Instance profiles*
- Amazon Machine Images (AMI)

When creating a new EC2 instance, or defining a launch configuration which will be used together with auto scaling groups, the AWS administrator can provide a script which will be run by the EC2 instance operating system as one of the last boot steps. This script, also called user data, is stored by AWS in the instance meta data and retrieved by the OS during boot. In Ubuntu the cloud-init daemon is responsible from retrieving and running this script.

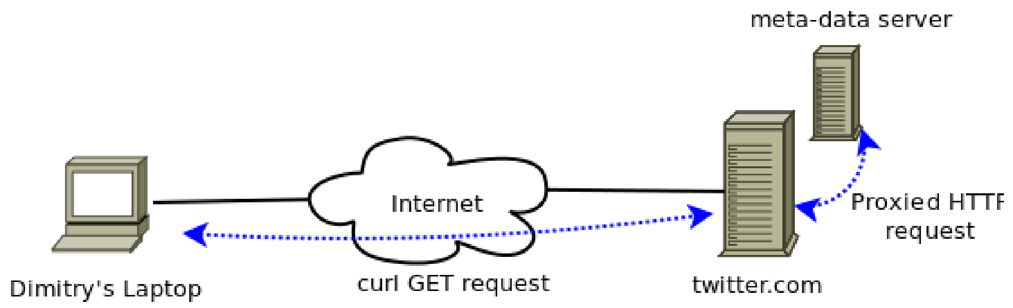
User data scripts are a common way to configure EC2 instances and their common structure follows these steps:

- Base package installation and updating
- Install Git client
- Define variables such as source code repository URL, branch and SSH keys
- Download application source code used in this instance from repository
- Compile and/or deploy the source code
- Start the required daemons

Since in most cases the repository where the instance's application source code is private, SSH keys are used to access it. GitHub, BitBucket and other widely used source repositories call these "Deploy SSH Keys". The SSH keys used to access the repository are usually hard-coded into the user data script, or stored in an alternate location where the script can download them.

This represents a risk when, because of a vulnerability, an attacker is able to proxy HTTP GET requests through the EC2 instance which allows him to retrieve the user data script from the meta data. In other words, if there is a way for the attacker to ask any of the services running on the instance to perform an HTTP GET request to arbitrary URLs and then return the HTTP response body to the attacker, then he would get access to the repository URL, branch and SSH keys allowing him to access the application source.

The most common vulnerability that allows this type of access is a PHP Remote File Include but any other vulnerable software which allows HTTP proxying could be used to retrieve the meta-data too.



Instance meta-data exposed via application vulnerability

The nimbostratus tool developed as part of this research has the capability to download all the instance meta data, including the user data script by exploiting a web application which is vulnerable to HTTP request proxying. The following console dump shows an example run:

```
dimitry@laptop:~/$ ./nimbostratus -v dump-ec2-metadata \
                   --mangle-function=core.utils.mangle.mangle
Starting dump-ec2-metadata
Request http://target.com/?url=http://169.254.169.254/latest/meta-data/
Request http://target.com/?url=http://169.254.169.254/latest/meta-data/instance-type
Request http://target.com/?url=http://169.254.169.254/latest/meta-data/instance-id
...
Instance type: t1.micro
AMI ID: ami-a02f66f2
Security groups: django_frontend_nimbostratus_sg
Availability zone: ap-southeast-1a
Architecture: x86_64
Private IP: 10.130.81.89
User data script was written to user-data.txt
```

It’s important to notice that in the previous run the target host (target.com), and the algorithm to extract the meta data information from a vulnerability present in that site is defined inside the source code of the `core.utils.mangle.mangle` function.

The source code if the `core.utils.mangle.mangle` can be adapted to exploit any vulnerability which allows HTTP request proxying, making the tool flexible to use in any environment.

Instance profiles

It is common practice for applications running on EC2 instances to access AWS services like SQS or S3. In order for this to work the application needs to have access to AWS credentials, there are various ways to achieve this, but Amazon AWS recommends using instance profiles.

Instance profiles are defined by the AWS architect, who defines which permissions will be available to the EC2 instances using the profile. For example, it is possible to create an instance profile with “SQS:*” permissions which would allow access to all API calls in the SQS service.

Once created, the instance profile is associated with an EC2 instance or a launch configuration. When the instances are started AWS creates a unique set of access key, a secret key, and security token and makes them available to the instance through its meta data. Most libraries which consume AWS services, such as boto, know how to retrieve the meta data credentials and use them to access the AWS services.

Since the instance profile credentials are stored in the meta-data, it suffers from the same risks as any other information stored there. The following is a run from one of the tools developed during this research which retrieves the instance profile credentials:

```
dimitry@laptop:~/$ ./nimbostratus -v dump-credentials \
                   --mangle-function=core.utils.mangle.mangle
Starting dump-credentials
Request http://target.com/?url=http://169.254.169.254/latest/.../security-credentials
Request http://target.com/?url=http://169.254.169.254/latest/.../django_sg
Found credentials
  Access key: ASIAJ5BQOUJRD4OPB4SQ
  Secret key: 73PUhbs7roCKP5zUEwUakH+49US4KTzp0j4oeuwF
  Token: AQo...OPzkAU=
```

Once those credentials are retrieved from the instance, it is possible to use them in any other system with Internet access. The permissions available to the attacker using the stolen credentials will be the same as the AWS EC2 instance, making it very important for the AWS administrator to use the least privilege principle for all AWS permissions. A example of how critical instance profile permissions are can be found in the *IAM:* privilege escalation* section.

Enumerating the permissions available to a set of credentials which was dumped from a remote EC2 instance might be challenging in cases where the profile doesn’t have permissions to access the IAM services. A tool was created to enumerate the permissions:

```
dimitry@laptop:~/$ ./nimbostratus -v dump-permissions --access-key ... \
                   --secret-key ... --token ...
Starting dump-permissions
...
{u'Statement': [{u'Action': ['ListQueues'],
                 u'Effect': u'Allow',
                 u'Resource': u'*'}],
 u'Version': u'2012-10-17'}
```

The tool will enumerate the permissions using different techniques, being bruteforce the last resource. The bruteforce approach will simply use the provided credentials to access different parts of Amazon’s AWS API and analyze the answer. This works very well but in order to avoid excessive run times and generating charges to the target infrastructure the tool only tests for a subset of the API calls, specifically the ones that get or list already existing resources.

IAM:* privilege escalation

Amazon’s IAM service is used to manage users, groups, roles and permissions. The permissions assigned to a group or user are fine grained and are usually created using Amazon’s IAM policy generator and then set using Amazon’s IAM service. An Amazon architect can create a custom permission set which would allow access to the different AWS services such as SQS, RDS, EC2 and IAM itself.

If special care is not taken by the AWS architect when assigning IAM permissions to a user, he could use IAM API calls to elevate his privileges. An example follows:

- AWS user Alice only has privileges to access IAM API calls, IAM:* for short
- Alice uses those privileges to create a new user: Bob
- Alice creates a new role with permissions to access all AWS services
- Alice assigns the newly created role to Bob
- Alice creates access keys for the user Bob

- Alice accesses any AWS service using Bob’s user

The permission set assigned to the user Bob would look like:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
```

To run this attack Alice requires at least these IAM permissions:

- CreateUser
- CreateAccessKey
- PutUserPolicy

It is important to notice that it would be also possible to achieve the same goal using other calls to the IAM service, for example it is possible to create a group, assign the policy to that group and then make the newly created user part of the group; or even make Alice part of the new group with high privileges.

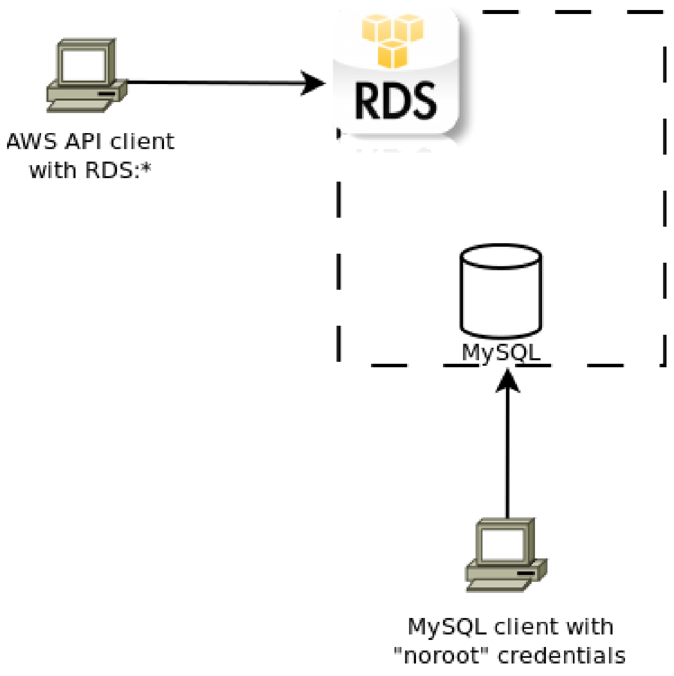
Using AWS to access virtualized database information

One of the most popular services provided by Amazon is RDS, which provides managed SQL databases. RDS reduces the management required by database servers and makes scaling and high availability easy to achieve.

SQL databases started from RDS can be managed using two very disctinctive methods:

- SQL database root user, connecting to the SQL server port (ie. 3306 in MySQL)
- Amazon’s RDS API, sending HTTPS requests to the RDS API endpoint

Each method allows the user to perform different actions on the database, information and users.



Different ways to access RDS and MySQL

Imagine the following situation:

- An intruder got access to a set of AWS credentials
- The credentials have permissions to access RDS:*
- The intruder has no other knowledge nor access to the SQL DB running on RDS

Any knowledgeable intruder will identify three API calls which could be used to access the information stored in SQL databases managed by RDS: CreateDBSnapshot, RestoreDBInstanceFromDBSnapshot and ModifyDBInstance. The steps are trivial:

- Use CreateDBSnapshot to create a backup of the RDS instance we want to get access to
- Use RestoreDBInstanceFromDBSnapshot to create a new RDS instance with all the information from the original one
- When the instance is running we'll still won't be able to access it using the SQL server port, since we don't have valid credentials for that. To solve that we call ModifyDBInstance, which will change the "root" user's password.
- Using a SQL client (ie. mysql-client in Ubuntu) the intruder can connect to the DB using the "root" user and the credentials set in ModifyDBInstance

Please note that an intruder could also have called ModifyDBInstance on an existing RDS instance and change the "root" password, which could be highly destructive and create a denial of service if the root user is used to access the SQL database from within the application, but also will grant him "root" account access to the SQL server.

Tools

Two tools were created as part of this research:

- nimbostratus: Tools to help with the enumeration and exploitation of AWS misconfigurations
- nimbostratus-target: Fabric based tool to spawn a vulnerable AWS environment where `nimbostratus` can be tested.

Both can be found by accessing the nimbostratus project site.

Building a secure AWS infrastructure

Use IAM instead of your root account

Amazon does a good job at recommending AWS users to use IAM generated users with fine grained permissions instead of using the root account credentials from within your EC2 instances. This is a good practice and should be followed all the time.

Different users for different tasks

Assign the least possible privilege for each of the instance profiles and users. Split the users into groups and manage fine-grained permissions for each.

Use instance profiles

Even with the risks mentioned above, instance profiles are (in the opinion of the writer) the safest and simplest way to provide AWS credentials to EC2 instances. The risks associated with other solutions such as hard-coding credentials in the (web) application source code are even higher than the ones instance profiles have.

In the future it would be nice to see meta-data information being migrated to a different delivery method / protocol which provides a higher degree of security, but that's only possible if Amazon changes their infrastructure.

Audit users and groups

Depending on the information security requirements of the Amazon infrastructure it might be a requirement to periodically audit the permissions assigned to each user.

Keeping an audit log of any modification of the users and profiles and sending email alerts based on specific rules could also be useful in some environments.

Summary

Cloud infrastructures are going mainstream and the lack of understanding of their internals by most part of the information security community will threathen their overall security level. This research exposed misconfigurations and vulnerabilities which could be exploited by intruders to get access to private information, impersonate EC2 instances, enumerate permissions and get access to all information related to the vulnerable infrastructure.

Better documentation and potentially a more secure implementation of the instance profile credential delivery method by Amazon would improve the overall security level of Amazon AWS deployments.

More research is required from the information security on this subject, I’m writing this paper hoping that others will venture into the clouds and learn as much as I did in the process.