Using ec2 instances as sneaker bid bots pt 1.

Date: 2023-11-21
By: csr13

Download as PDF

Use case, you have produced some program that executes on an online shop when certain item "drops" release, the store is very popular and you want to get as many items you want so you can resell them for 50% profit on your online store -- I am talking about for example, Nike Sneakers, which drop and have a lifetime of about 4 hours before they sell out.

Well one way of ensuring you obtain these products is if you have the release date of the product and you deposit them on a database table or a cvs file and have cron jobs reading the file every day and checking the release date and executing an order to spaw 100+ bots per product release(ec2 instances with different ip addresses using a proxy to make a purchase, so websites like nike, supreme and adidas won't block you), the cost of doing this is about .008 cents per ec2 instance spawned, because the order is to spaw, execute and shutdown.

Usually you want to have experience with the following to follow along.

- Aws console
- Creating VPN (Virtual Private Networks)
- Creating instances (Machines inside cloud private network)
- Creating snapshop of an initial machine (to clone bots off this machine exact build)
- Python and using boto library for connecting with AWS, and django web framework
- AWS Secrets Manager

Let's say that you already have the main origin ec2, and ec2 is a machine on the cloud living in your virtual private network.

If you have that then you also have the following.

- AMI ID
- SUBNET ID
- SECURITY GROUP ID
- VPC ID
- NETWORK INTERFACE ID
- INTERNET GATEWAY

Now you need to store these values secureley, they can't live in your code because it's just not safe to do so, so I use this Loader that I wrote to pull secrets from AWS Secrets Manager services.

```python
import os
import logging

from django.core.exceptions import ImproperlyConfigured
import boto3


logger = logging.getLogger(__name__)


class AwsSecretLoader(object):
    @classmethod
    def _log(cls, e):
        if e.response["Error"]["Code"] == "ResourceNotFoundException":
            logger.error(f"The requested secret { e } was not found")
        elif e.response["Error"]["Code"] == "InvalidRequestException":
            logger.error("The request was invalid.")
        elif e.response["Error"]["Code"] == "InvalidParameterException":
            logger.error("The request had invalid params")
        elif e.response["Error"]["Code"] == "DecryptionFailure":
            logger.error(
                "The requested secret can't be decrypted using the provided KMS key"
            )
        elif e.response["Error"]["Code"] == "InternalServiceError":
            logger.error("Fatal error")

    @classmethod
    def get_aws_secret(cls, name):

        region_name = os.getenv("AWS_REGION_NAME", "us-east-1")
        session = boto3.session.Session()
        client = session.client(
            service_name="secretsmanager",
            region_name=region_name,
        )

        try:
            get_secret_value_response = client.get_secret_value(SecretId=name)
        except Exception as e:
            error = f"{str(e)} [SECRET_NAME] = {name}"
            raise ImproperlyConfigured(error)

        if "SecretString" in get_secret_value_response:
            data = get_secret_value_response["SecretString"]
        else:
            data = get_secret_value_response["SecretBinary"]

        return data
```

Ok, this `Loader` object lives in `loader.py`

Now Tests so is possible to test the functionality before writing any code inside the main client codebase, testing oriented programming is like streching pre working out, some people do it, others don't but it is very beneficial if you do so, because your preformance, during working (or program execution) is better.

You will need an ec2 `init` script, which will be the first thing any ec2 clones will execute after being created on demand, this is what I may look like

```
INIT= '''Content-Type: multipart/mixed; boundary= "//"
MIME-Version: 1.0

--//
Content-Type: text/cloud-config; charset= "us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename= "cloud-config.txt"

#cloud-config
cloud_final_modules:
    - [scripts-user, always]

--//
Content-Type: text/x-shellscript; charset= "us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename= "userdata.txt"

#!/bin/bash
systemctl restart checkout-service.service
--//'''
```

All this does is it instructs the ec2 instance to execute `systemctl restart checkout-service.service` which is the program that the ec2 executes to make the product purchase.

Now I want to test starting ec2 instances programatically with this init script via a module that my web application will run to trigger this mass action.

```
import json
import logging
import time

import boto3
from django.conf import settings
from django.test import TestCase
from aws.loader import AwsSecretLoader
from aws.models import UserAwsCheckoutResource
from users.models import User
from notifications.models import UserNotification


LOADER= AwsSecretLoader()

logger= logging.getLogger(__name__)

# ---------------------------------------------------
# Identifiers of main network components.
# ---------------------------------------------------

CHECKOUT_AMI_ID= json.loads(
    LOADER.get_aws_secret(
        "CHECKOUT_AMI_ID"
    )
)["CHECKOUT_AMI_ID"]


CHECKOUT_SUBNET_ID= json.loads(
    LOADER.get_aws_secret(
        "CHECKOUT_SUBNET_ID"
    )
```

```python
)["CHECKOUT_SUBNET_ID"]


CHECKOUT_SECURITY_GROUP_ID= json.loads(
    LOADER.get_aws_secret(
        "CHECKOUT_SECURITY_GROUP_ID"
    )
)["CHECKOUT_SECURITY_GROUP_ID"]


CHECKOUT_VPC_ID= json.loads(
    LOADER.get_aws_secret(
        "CHECKOUT_VPC_ID"
    )
)["CHECKOUT_VPC_ID"]


CHECKOUT_NETWORK_INTERFACE_ID= json.loads(
    LOADER.get_aws_secret(
        "CHECKOUT_NETWORK_INTERFACE_ID"
    )
)["NETWORK_INTERFACE_ID"]


CHECKOUT_INTERNET_GATEWAY_ID= json.loads(
    LOADER.get_aws_secret(
        "CHECKOUT_INTERNET_GATEWAY"
    )
)["CHECKOUT_INTERNET_GATEWAY"]


# ----------------------------------------------------
# Use boto3.client to create things on aws
# ----------------------------------------------------


EC2_RESOURCE= boto3.resource('ec2')
EC2_CLIENT= boto3.client('ec2')

INIT= '''Content-Type: multipart/mixed; boundary= "//"
MIME-Version: 1.0

--//
Content-Type: text/cloud-config; charset= "us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename= "cloud-config.txt"

#cloud-config
cloud_final_modules:
    - [scripts-user, always]

--//
Content-Type: text/x-shellscript; charset= "us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename= "userdata.txt"

#!/bin/bash
systemctl restart checkout-service.service
--//'''


class TestEc2Wrapper(TestCase):
    fixtures= ["user-fixtures.json"]

    def setUp(self):
        self.resource= EC2_RESOURCE
        self.client= EC2_CLIENT
        self.user= User.objects.get(email= "admin@admin.com")

    def test_get_instance_(self):

        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        # This will get called when the user subscribes and pays actual money.
        # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        # Make the instances.
        # ~~~~~~~~~~~~~~~~~~~

        instances= self.resource.create_instances(
            ImageId="ami-027ece036eecc0eea",
            MaxCount=1,
            MinCount=1,
            InstanceType="t2.micro",
            TagSpecifications=[
                {
                    "ResourceType": "instance",
                    "Tags": [
                        {
```

```
                            "Key": "USER",
                            "Value": self.user.email
                        }
                    ],
                },
            ],
            KeyName="puppeeter",
            NetworkInterfaces=[
                {
                    'SubnetId': CHECKOUT_SUBNET_ID,
                    'DeviceIndex': 0,
                    'AssociatePublicIpAddress': True,
                    'Groups': [
                        CHECKOUT_SECURITY_GROUP_ID,
                    ]
                }
            ],
            UserData=INIT
        )

        for instance in instances:
            UserAwsCheckoutResource.cook_user_aws_checkout_resource(
                **dict(
                    user=self.user,
                    instance_id=instance.id,
                    public_ip="-",
                    meta=instance.meta.data,
                )
            )

        # ============================
        # Optimzing this is a priority
        # ============================

        iterator=0
        while iterator <= 100:

            current_instances=[]
            for instance in instances:
                current_instances.append(
                    self.resource.Instance(id=instance.id)
                )

            states_and_ids=[]
            for each in current_instances:
                states_and_ids.append(
                    [
                        each.state['Name'],
                        each.id
                    ]
                )

            if not all(
                list(
                    map(
                        lambda x: x[0] == "running", states_and_ids
                    )
                )
            ):
                time.sleep(6)
                iterator += 1
                continue

            for each in current_instances:
                each.stop()

            for inst in current_instances:

                # ===================================
                # Get the IP address of the instance.
                # ===================================

                try:
                    ip=inst.public_ip_address
                    if ip is None or ip == "":
                        ip=inst.private_ip_address
                        if ip is None or ip == "":
                            ip="-"
                except Exception as error:
                    logger.error(str(error))
                    ip="-"

                try:
                    user_instance=UserAwsCheckoutResource.objects.get(
                        user=self.user,
                        instance_id=inst,
                        public_ip=ip
                    )
                except Exception as e:
                    error = e.args
                    continue

                # ====================================
                # Save the new state and the ip to the
                # user instance resource
                # ====================================

                user_instance.is_active=False
                user_instance.public_ip=ip
                user_instance.save()

            break

        # ==========================================
        # Send the user a notification of this event.
        # ==========================================

        bid=1
        UserNotification.objects.create(
            user=self.user,
            data=json.dumps(
                {
                    "status": "success",
                    "type": settings.EVENT_TYPES["RESOURCE_CREATED"],
                    "message": f"Proxy created for bid {bid}"
                }
            )
        )
```

I have added comments on the code so it is redable, but I just want to notice, the instances are started, and the number of instances are defined by some parameters when created, then this instances are related to a user, stored on the database, and on AWS these instances have the user username appended to their id, so they can be queried, regardless I store the ids on local database to keep track of which instances which user creates, their status, and more, for bookeeping.

Only after instances are all up and running I can extract their public ip, to give them the purchase command on my main web application via an RPC (remote procedure call) and kind of have a command and control webserver, where I can control and instruct purchase bots, because I have each ec2 instance public ip, their id stored on my db. Therfore, because the init script boots my listener `checkout.service` program, via init command, which has RPC endpoints to trigger purchase and handle purchase errors, as well as other actions that I can use to relay tasks from my main C2 (command and control server) back to my botnet or purchase ec2 instances.

Well well, now what ... well the good part has just started, now, it is time to make this a product people can use, and integrate some payment options via stripe so people can place bets on the sneakers they want, let's say new AIR JORDANS dropping soon, well I list them on my .onion site (tor site) then have a client invitation only server that has custom login/dashboard, have payment gateway so I can charge them the price for the sneakers, plus 2.00 USD per ec2 instance, which increases their chance to obtain their purchase (well, let's say the ec2 instance costs me 0.008 cents, and I am charging 2.00 usd per ec2 instance, each homie pays 15 usd of instances I am not even paying .50 cents of ec2 instances)

Part 2 dropping soon, this will include, backend API's to handle frontend client dashboard purchase orders, ec2 creation webhooks from success payment gateway (post purchase) which triggers the code we tested, as well as how I instruct the ec2 instances the client payd for to execute their purchase of the model they want, and how refunds are handled if any error occured, because well shit happens.

This post is part of a series, check out the other parts of this series of notes

Using ec2 instances as sneaker bid bots pt 2.

Using ec2 instances as sneaker bid bots pt 3.

Related Notes

1) Using ec2 instances as sneaker bid bots pt 2.
Download PDF
Date published: 2023-11-27
bots python aws series

2) Whatsapp chatbot with Python and Twilio
Download PDF
Date published: 2023-11-15
bots python whatsapp business

3) Using ec2 instances as sneaker bid bots pt 3.
Download PDF
Date published: 2023-11-28
bots python aws series

4) Real Time Language Translation Agent System for Call Centers
Download PDF

Date published: 2023-11-16
voip telephony python systems

5) Using ec2 instances as sneaker bid bots pt 2.
Download PDF
Date published: 2023-11-27
bots python aws series

6) Whatsapp chatbot with Python and Twilio
Download PDF
Date published: 2023-11-15
bots python whatsapp business

7) Backend Celery task manager dashboard via Flower
Download PDF
Date published: 2023-10-29
backend tasks python

8) Using ec2 instances as sneaker bid bots pt 3.
Download PDF
Date published: 2023-11-28
bots python aws series

9) Using ec2 instances as sneaker bid bots pt 2.
Download PDF
Date published: 2023-11-27
bots python aws series

10) Using ec2 instances as sneaker bid bots pt 3.
Download PDF
Date published: 2023-11-28
bots python aws series

11) Using ec2 instances as sneaker bid bots pt 2.
Download PDF
Date published: 2023-11-27
bots python aws series

12) Using ec2 instances as sneaker bid bots pt 3.
Download PDF
Date published: 2023-11-28
bots python aws series