AWS IR Documentation

Release 0.3.0

Joel Ferrier, Andrew Krug

Jul 08, 2018

Contents

1 Quickstart **3**
- 1.1 Installation . . . 3
- 1.2 AWS Credentials . . . 3
- 1.3 Setup Roles with Cloudformation . . . 3
- 1.4 Key Compromise . . . 4
- 1.5 Instance Compromise . . . 4

2 Installation **7**
- 2.1 System Requirements . . . 7
- 2.2 Installing from PyPi . . . 7
- 2.3 Installing From Github . . . 7
- 2.4 Local Build and Install . . . 7
- 2.5 Local Execution . . . 8
- 2.6 Using Docker . . . 8
- 2.7 AWS Credentials Using MFA and AssumeRole . . . 8
- 2.8 Some Linux distributions require additional system packages . . . 8

3 Development **11**
- 3.1 Types of Development . . . 11
- 3.2 Plugins . . . 11
- 3.3 CLI Development . . . 16

4 About **17**
- 4.1 License . . . 17

Python installable command line utility for mitigation of host and key compromises.

CHAPTER 1

Quickstart

First, *Install aws_ir*.

1.1 Installation

```
$ python3 -m virtualenv env
$ source/env/bin/activate
$ pip install aws_ir
```

For other installation options see: installing.

1.2 AWS Credentials

Ensure aws credentials are configured under the user running aws_ir as documented by amazon.

1.3 Setup Roles with Cloudformation

A cloudformation stack has been provided to setup a group and a responder role.

Simply create the stack available at:

https://github.com/ThreatResponse/aws_ir/blob/master/cloudformation/responder-role.yml.

Then add all your users to the IncidentResponders group. After that you're good to go!

Note that this roles has a constraint that all your responders use MFA. .. code-block:: bash

aws:MultiFactorAuthPresent: 'true'

1.4 Key Compromise

The `aws_ir` subcommand `key-compromise` disables access keys in the case of a key compromise. It's single argument is the access key id, he compromised key is disabled via the AWS api.

```
usage: aws_ir key-compromise [-h] --access-key-id ACCESS_KEY_ID
                             [--plugins PLUGINS]

optional arguments:
  -h, --help            show this help message and exit
  --access-key-id ACCESS_KEY_ID
  --plugins PLUGINS     Run some or all of the plugins in a custom order.
                        Provided as a comma separated listSupported plugins:
                        disableaccess_key,revokests_key
```

Below is the output of running the `key-compromise` subcommand.

```
$ aws_ir key-compromise --access-key-id AKIAINLHPIG64YJXPK5A
2017-07-20T21:04:01 - aws_ir.cli - INFO - Initialization successful proceeding to␣
↪incident plan.
2017-07-20T21:04:01 - aws_ir.plans.key - INFO - Attempting key disable.
2017-07-20T21:04:03 - aws_ir.plans.key - INFO - STS Tokens revoked issued prior to␣
↪NOW.
2017-07-20T21:04:03 - aws_ir.plans.key - INFO - Disable complete.  Uploading results.
Processing complete for cr-17-072104-7d5f
Artifacts stored in s3://cloud-response-9cabd252416b4e5a893395c533f340b7
```

1.5 Instance Compromise

The `aws_ir` subcommand `instance-compromise` preserves forensic artifacts from a compromised instance after isolating the instance. Once all artifacts are collected and tagged the compromised instance is powered off. The `instance-compromise` subcommand takes three arguments, the `instance-ip` of the compromised instance, a `user` with ssh access to the target instance, and the `ssh-key` used for authentication.

Currently `user` must be capable of passwordless sudo for memory capture to complete. If `user` does not have passwordless sudo capabilities all artifiacts save for the memory capture will be gathered.

Note: AWS IR saves all forensic artifacts except for disk snapshots in an s3 bucket created for each case. Disk snapshots are tagged with the same case number as the rest of the rest of the artifacts.

Below is the output of running the `instance-compromise` subcommand.

```
$  aws_ir --examiner-cidr-range '4.4.4.4/32' instance-compromise --target 52.40.162.
↪126 --user ec2-user --ssh-key ~/Downloads/testing-041.pem
   2017-07-20T21:10:50 - aws_ir.cli - INFO - Initialization successful proceeding to␣
↪incident plan.
   2017-07-20T21:10:50 - aws_ir.libs.case - INFO - Initial connection to␣
↪AmazonWebServices made.
   2017-07-20T21:11:03 - aws_ir.libs.case - INFO - Inventory AWS Regions Complete 14␣
↪found.
   2017-07-20T21:11:03 - aws_ir.libs.case - INFO - Inventory Availability Zones␣
↪Complete 37 found.
   2017-07-20T21:11:03 - aws_ir.libs.case - INFO - Beginning inventory of resources␣
↪world wide.  This might take a minute...
```

(continues on next page)

(continued from previous page)

```
  2017-07-20T21:11:03 - aws_ir.libs.inventory - INFO - Searching ap-south-1 for␣
↪instance.
  2017-07-20T21:11:05 - aws_ir.libs.inventory - INFO - Searching eu-west-2 for␣
↪instance.
  2017-07-20T21:11:05 - aws_ir.libs.inventory - INFO - Searching eu-west-1 for␣
↪instance.
  2017-07-20T21:11:06 - aws_ir.libs.inventory - INFO - Searching ap-northeast-2 for␣
↪instance.
  2017-07-20T21:11:07 - aws_ir.libs.inventory - INFO - Searching ap-northeast-1 for␣
↪instance.
  2017-07-20T21:11:08 - aws_ir.libs.inventory - INFO - Searching sa-east-1 for␣
↪instance.
  2017-07-20T21:11:09 - aws_ir.libs.inventory - INFO - Searching ca-central-1 for␣
↪instance.
  2017-07-20T21:11:09 - aws_ir.libs.inventory - INFO - Searching ap-southeast-1 for␣
↪instance.
  2017-07-20T21:11:10 - aws_ir.libs.inventory - INFO - Searching ap-southeast-2 for␣
↪instance.
  2017-07-20T21:11:11 - aws_ir.libs.inventory - INFO - Searching eu-central-1 for␣
↪instance.
  2017-07-20T21:11:12 - aws_ir.libs.inventory - INFO - Searching us-east-1 for␣
↪instance.
  2017-07-20T21:11:13 - aws_ir.libs.inventory - INFO - Searching us-east-2 for␣
↪instance.
  2017-07-20T21:11:13 - aws_ir.libs.inventory - INFO - Searching us-west-1 for␣
↪instance.
  2017-07-20T21:11:13 - aws_ir.libs.inventory - INFO - Searching us-west-2 for␣
↪instance.
  2017-07-20T21:11:14 - aws_ir.libs.case - INFO - Inventory complete.  Proceeding to␣
↪resource identification.
  2017-07-20T21:11:14 - aws_ir.plans.host - INFO - Proceeding with incident plan␣
↪steps included are ['gather_host', 'isolate_host', 'tag_host', 'snapshotdisks_host',
↪ 'examineracl_host', 'get_memory', 'stop_host']
  2017-07-20T21:11:14 - aws_ir.plans.host - INFO - Executing step gather_host.
  2017-07-20T21:11:15 - aws_ir.plans.host - INFO - Executing step isolate_host.
  2017-07-20T21:11:16 - aws_ir.plans.host - INFO - Executing step tag_host.
  2017-07-20T21:11:17 - aws_ir.plans.host - INFO - Executing step snapshotdisks_host.
  2017-07-20T21:11:17 - aws_ir.plans.host - INFO - Executing step examineracl_host.
  2017-07-20T21:11:19 - aws_ir.plans.host - INFO - Executing step get_memory.
  2017-07-20T21:11:19 - aws_ir.plans.host - INFO - attempting memory run
  2017-07-20T21:11:19 - aws_ir.plans.host - INFO - Attempting run margarita shotgun␣
↪for ec2-user on 52.40.162.126 with /Users/akrug/Downloads/testing-041.pem
  2017-07-20T21:11:21 - margaritashotgun.repository - INFO - downloading https://
↪threatresponse-lime-modules.s3.amazonaws.com/modules/lime-4.9.32-15.41.amzn1.x86_64.
↪ko as lime-2017-07-21T04:11:21-4.9.32-15.41.amzn1.x86_64.ko
  2017-07-20T21:11:25 - margaritashotgun.memory - INFO - 52.40.162.126: dumping␣
↪memory to s3://cloud-response-a0f2d7e68ef44c36a79ccfe4dcef205a/52.40.162.126-2017-
↪07-21T04:11:19-mem.lime
  2017-07-20T21:15:43 - margaritashotgun.memory - INFO - 52.40.162.126: capture 10%␣
↪complete
  2017-07-20T21:19:37 - margaritashotgun.memory - INFO - 52.40.162.126: capture 20%␣
↪complete
  2017-07-20T21:23:41 - margaritashotgun.memory - INFO - 52.40.162.126: capture 30%␣
↪complete
  2017-07-20T21:28:17 - margaritashotgun.memory - INFO - 52.40.162.126: capture 40%␣
↪complete
  2017-07-20T21:32:42 - margaritashotgun.memory - INFO - 52.40.162.126: capture 50%␣
↪complete
```

(continues on next page)

(continued from previous page)

```
   2017-07-20T21:37:18 - margaritashotgun.memory - INFO - 52.40.162.126: capture 60%␣
↪complete
   2017-07-20T21:39:18 - margaritashotgun.memory - INFO - 52.40.162.126: capture 70%␣
↪complete
   2017-07-20T22:00:13 - margaritashotgun.memory - INFO - 52.40.162.126: capture 80%␣
↪complete
   2017-07-20T22:04:19 - margaritashotgun.memory - INFO - 52.40.162.126: capture 90%␣
↪complete
   2017-07-20T22:17:32 - margaritashotgun.memory - INFO - 52.40.162.126: capture 100%␣
↪complete
   2017-07-20T21:41:52 - aws_ir.plans.host - INFO - memory capture completed for: [
↪'52.40.162.126'], failed for: []
   2017-07-20T21:41:52 - aws_ir.plans.host - INFO - Executing step stop_host.

Processing complete for cr-17-072104-7d5f
Artifacts stored in s3://cloud-response-a0f2d7e68ef44c36a79ccfe4dcef205a
```

Note that `aws_ir instance-compromise` installs margarita shotgun on your local machine to perform memory capture. Doing so requires trusting the GPG key of security@threatresponse.cloud, which can be done with the command:

```
$ curl -s https://threatresponse-lime-modules.s3.amazonaws.com/REPO_SIGNING_KEY.asc |␣
↪gpg --import -
gpg: key 67172B17: public key "Lime Signing Key (Threat Response Official Lime␣
↪Signing Key) <security@threatresponse.cloud>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

CHAPTER 2

Installation

2.1 System Requirements

ThreatResponse requires python >= 3.4.

2.2 Installing from PyPi

2.3 Installing From Github

```
$ python3 -m virtualenv env
$ source/env/bin/activate
$ pip install git+ssh://git@github.com/ThreatResponse/aws_ir.git@master
$ aws_ir -h
```

2.4 Local Build and Install

```
$ git clone https://github.com/ThreatResponse/aws_ir.git
$ cd aws_ir
$ python3 -m virtualenv env
$ source/env/bin/activate
$ pip install .
$ aws_ir -h
```

2.5 Local Execution

In the previous two example dependencies are automatically resolved, if you simply want to run aws_ir using the script `bin/aws_ir` you will have to manually install dependencies

```
$ git clone https://github.com/ThreatResponse/aws_ir.git
$ cd aws_ir
$ python3 -m virtualenv env
$ source/env/bin/activate
$ pip install -r requirements.txt
$ ./bin/aws_ir -h
```

2.6 Using Docker

```
$ git clone https://github.com/ThreatResponse/aws_ir.git
$ cd aws_ir
$ docker-compose build aws_ir
$ docker-compose run aws_ir bash
$ pip install .
```

2.7 AWS Credentials Using MFA and AssumeRole

Many users of aws_ir have requested the ability to use the tooling with mfa and assumeRole functionality. While we don't natively support this yet v0.3.0 sets the stage to do this natively by switching to boto-session instead of thick clients.

For now if you need to use the tool with MFA we recommend:

https://pypi.python.org/pypi/awsmfa/0.2.4.

```
aws-mfa \
--device arn:aws:iam::12345678:mfa/bobert \
-assume-role arn:aws:iam::12345678:role/ResponderRole \
--role-session-name \"bobert-ir-session\"
```

awsmfa takes a set of long lived access keys from a boto profile called [default-long-lived] and uses those to generate temporary session tokens that are automatically put into the default boto profile. This ensures that any native tooling that doesn't support MFA + AssumeRole can still leverage MFA and short lived credentials for access.

2.8 Some Linux distributions require additional system packages

2.8.1 Fedora / RHEL Distributions

- python-devel (Python 3.4+)
- python-pip
- libffi-devel
- libssl-devel

2.8.2 Debian Distributions

- python-dev (Python 3.4+)
- python-pip
- libffi-dev
- libssl-dev

CHAPTER 3

Development

Congratulations on taking the first step to become a developer on aws_ir! We're a very un-opinionated and forgiving community to work in. This guide will cover two different types of development for the aws_ir command line interface.

3.1 Types of Development

1. Plugins *(These are the actual incident steps.)*
2. The CLI itself

3.2 Plugins

Plugins are probably the easiest way to get started as a developer. Since v0.3.0 the command line interface now supports dynamically loading plugins from source using a python module called PluginBase.

3.2.1 Getting Started

First create a folder in your home directory called .awsir. This is automatically searched each time awsir is run. *Warning: If you put python code in here that can not be executed it will prevent your command line from running.*

```
$ mkdir ~/.awsir
```

Excellent! You are well on your way to creating you first plugin.

3.2.2 Naming your plugin

We prefer descriptive names based on the type of resource that will be interacted with. Currently aws_ir supports:

- Host Compromises
- Key Compromises
- Lambda Compromises (*Coming Soon*)
- Role Compromises (*Coming Soon*)

The *TLDR* here is to name your plugin following the standard:

- THETHINGITDOES_key.py
- THETHINGITDOES_host.py
- THETHINGITDOES_lambda.py
- THETHINGITDOES_role.py

Let's start a new plugin and we'll call it foo_key.py.

```
$ touch ~/.awsir/foo_key.py
```

3.2.3 Plugin Boilerplate

Inside of that file foo_key.py there is some minimum content that has to exist just to get started. All plugins follow a standard object pattern or they would not be plugins.

```
import logging

# Initializing the stream logger here ensures that any logger messages
# bubble up into the case logs from the plugin.

logger = logging.getLogger(__name__)

class Plugin(object):
  def __init__(
      self,
      boto_session,
      compromised_resource,
      dry_run
  ):

      # AWS_IR generates a boto session that is handed off to each plugin.
      # This ensures as a developer you can create boto3 resource or client.
      self.session = boto_session

      # The compromised resource also gets handed of to the plugin.
      # This is slightly different depending on whether this is a
      # host of key resource.
      # See: https://github.com/ThreatResponse/aws_ir/blob/master/aws_ir/
↪libs/compromised.py
      self.compromised_resource = compromised_resource

      # Each incident plan also sends through the type of compromise.
      # key, host, lambda, etc.
      self.compromise_type = compromised_resource['compromise_type']
      self.dry_run = dry_run

      self.access_key_id = self.compromised_resource['access_key_id']
```

(continues on next page)

(continued from previous page)

```
    # The setup function should call any other private methods on the
    # object in order to achieve your IR step.  This facilitates easy
    # testing using PyUnit or PyTest.
    self.setup()

def setup(self):
    """Method runs the plugin."""
    if self.dry_run is not True:
      # The stuff we can not dry run goes here.
      self._your_private_step()
      self._your_other_private_step()
    else:
      pass

def validate(self):
    """Returns whether this plugin does what it claims to have done"""
    pass

def output(self):
    """
    Future function that will be required of all plugins.  Will be
    required to contain a json schema validated payload to report on
    steps taken and assets generated.
    """
    pass

def _your_private_step(self):
    """Something you might do as part of IR."""

    # This is how to log a status message.
    logger.info("I just secured all the things!.")
    pass

def _your_other_private_step(self):
  """Something other thing you might do as part of IR."""
    pass
```

Those are the minimum required methods. Everything you decide to do after that in your aws_ir plugin is up to you. As log as Plugin() is initalized, validate is called, and output can be called the plugin will execute in the pipeline.

3.2.4 Considerations

You may want to get familiar with how boto sessions become boto3 resources and clients as a part of your training. This is well documented.

https://boto3.readthedocs.io/en/latest/reference/core/session.html.

You might also want to borrow our code or pull request an aws_ir core plugin into mainstream. We would love it if you were excited enough to do that.

All of our plugins install from this repository: https://github.com/ThreatResponse/aws_ir_plugins.

3.2.5 Host Compromised Resource

The host compromised resource is a little bigger than an access key since we need to store more information to do things like interact with the VPC. It's dictionary looks like this:

```
"compromised_resource" : {
  "public_ip_address": "4.2.2.2",
  "private_ip_address": "10.10.10.1",
  "instance_id": "i-xxxxxxxxxxxx",
  "launch_time": "DATETIME",
  "platform": "windows",
  "vpc_id": "vpc-xxxxxx",
  "ami_id": "ami-xxxxxx",
  "volume_ids": [
    "BlockDeviceMappings": []
  ],
  "region": "region"
}
```

Of course your can always just print(compromised_resource) while you're developing.

3.2.6 Testing your plugin

There are two primary ways to test your plugin. You can use the cli and actually run it against an instance or key. Or you can write a pyUnit test.

Testing with the CLI

1. Run the aws_ir cli help to see if your plugin is loading.

```
$ aws_ir instance-compromise --help
usage: aws_ir instance-compromise [-h] [--target TARGET] [--targets TARGETS]
                                  [--user USER] [--ssh-key SSH_KEY]
                                  [--plugins PLUGINS]

optional arguments:
  -h, --help          show this help message and exit
  --target TARGET     instance-id|instance-ip
  --targets TARGETS   File of resources to process instance-id or ip-address.
  --user USER         this is the privileged ssh user for acquiring memory from
                      the instance. Required for memory only.
  --ssh-key SSH_KEY   provide the path to the ssh private key for the user.
                      Required for memory only.
  --plugins PLUGINS   Run some or all of the plugins in a custom order.
                      Provided as a comma separated list of supported plugins:
                      examineracl_host,foo_host,gather_host,isolate_host,snapsh
                      otdisks_host,stop_host,tag_host,get_memory
```

If you see it in the list of plugins then it's getting picked up by the plugin loader and you can tell the cli to run only that plugin instead of a standard incident plan. *Note: foo_host in the above output.*

Testing with PyUnit

If you're familiar with PyUnit you can use spulec/moto and pyUnit to test your plugin prior to running in the CLI. We do this for aws_ir_plugins using TravisCI.

```
# Test boilerplate for an EC2 plugin
import boto3
import unittest

from aws_ir_plugins import sample_host
from moto import mock_ec2


class BoilerPlateTest(unittest.TestCase):
    # Begin mocking
    @mock_ec2
    def test_tag_host(self):
        # Create fake EC2 clients and sessions
        self.ec2 = boto3.client('ec2', region_name='us-west-2')
        session = boto3.Session(region_name='us-west-2')

        # Create a fake instance to process
        ec2_resource = session.resource('ec2')

        instance = ec2_resource.create_instances(
            ImageId='foo',
            MinCount=1,
            MaxCount=1,
            InstanceType='t2.medium',
            KeyName='akrug-key',
        )

        # Fake a compromised resource with the minimum set of fields needed
        self.compromised_resource = {
            'case_number': '123456',
            'instance_id': instance[0].id,
            'compromise_type': 'host'
        }

        # Execute the plugin
        plugin = sample_host.Plugin(
            boto_session=session,
            compromised_resource=self.compromised_resource,
            dry_run=False
        )

        result = plugin.validate()

        # Your test assertions

        assert result is True
```

I prefer to run these using nose and nose-watch during active development. Moto ensures that you're mocking all the EC2 calls so you can develop the plugin without effecting your AWS environment.

Example

```
nosetest --with-watch tests/test_sample.py
```

This is like guard in rails. It watches the file system and re-runs the test each time you write some code and save.

3.3 CLI Development

We are currently accepting pull requests for the aws_ir cli for features and bug fixes.

In order to develop the cli you will need to setup a python3 virtual environment. However, you'll need to start by cloning the code.

3.3.1 Pulling down the code and getting started

Step 1. Fork us on Github.

```
# Clone your fork

# 1. git clone
git@github.com:<your github here>/aws_ir.git
```

Step 2. Setup

2. setup a virtualenv (must be python3) cd aws_ir python3 -m virtualenv env

3. activate the virtualenv source env/bin/activate

4a. with setuptools pip install -e . python setup.py test python setup.py pytest –addopts='tests/test_cli.py'

—or –

4b. with local plugins and pytest-watch point requirements.txt to the local version of aws_ir_plugins *-e ../aws_ir_plugins* .. code-block:: bash

pip3 install -r requirements.txt ./bin/aws_ir -h ptw –runner "python setup.py test"

—or –

#4c. Use the docker container .. code-block:: bash

docker-compose build aws_ir docker-compose run aws_ir bash pip install -e .

Step 3. Develop!

Note: There is a helper script in *bin/aws_ir* that can be called to execute aws_ir.

When your feature is finished simply open a PR back to us.

If you have any questions please do file a github issue or e-mail info@threatresponse.cloud .

3.3.2 Using testpypi

To use a test build of aws_ir_plugins: in setup.py: - point the required version at aws_ir_plugins==0.0.3b123 (substitute the build you want) - add: dependency_links=['https://test.pypi.org/simple/aws-ir-plugins/']

CHAPTER 4

About

AWS IR is a part of the Threat Response project.

4.1 License

AWS IR is distributed under the MIT License (MIT).