Extending OpenC2 to the Cloud  
HACS498 project

Brian Cherin, Melissa Hoff, Rahul Kiefer, Balaji Kodali,  
Amber Owens, Selena Xiao, Andrew Schuck

Background
Open Command and Control (OpenC2) is a standardized language defined to support or administer cyber defense technologies. It was developed by a technical committee from the Organization for the Advancement of Structured Information Standards (OASIS) in 2019 with the purpose of being vendor and application agnostic. This enables computers and/or software to communicate with each other across a wide range of tools and applications designed for cybersecurity. With this interoperability of different tools, OpenC2 will decrease the incident response time and human error during security incidents and attacks. The standardization of interfaces and protocols through OpenC2 will allow security professionals of any organization to conduct automated threat responses across multiple cyber defense tools at the same time. Due to the broad nature of OpenC2, it has the capability to support future devices and tools along with existing technologies. This grants OpenC2 flexibility in a vast number of scenarios including those that do not even exist yet (OASIS Technical Committee, 2021).

OpenC2 and the Cloud
Cloud services are an integral part of the world economy, and due to Covid-19, individuals and businesses are reliant on cloud computing, storage, and applications more than ever. However, OpenC2 currently does not support cloud operations, and with the popularity of cloud services rising, our team wanted to research ways to extend the OpenC2 standard to the cloud. Below is a list of statistics from Radoslav Ch. illustrating how prominent cloud services are, and why businesses are rushing to transition on to “the cloud” (Ch., 2021):

- The average person uses 36 cloud-based services every single day
- 67% of enterprise infrastructure is cloud-based
- In 2020 a staggering 83% of the companies’ workload was stored on the cloud
- 80% of companies report operation improvements within the first few months of adopting to the cloud

The reason for these significant operational improvements is attributed to the cloud’s flexibility, minimal required maintenance, and ease of access. This delegation of responsibility from the company to the cloud provider (e.g. AWS, Microsoft Azure, etc.) allows the company to not worry about maintaining the hardware (both computational and storage), and instead they can quickly grow their business without worrying about physical machine configuration and costs. Amazon Web Services (AWS) provides a list of advantages of cloud computing (“Six Advantages of Cloud Computing,” n.d):
- **Trade capital expense for variable expense** - Instead of initially purchasing and investing in data centers, “you can pay only when you consume computing resources, and pay only for how much you consume”

- **Benefit from massive economies of scale** - “Because usage from hundreds of thousands of customers is aggregated in the cloud, providers such as AWS can achieve higher economies of scale, which translates into lower pay as-you-go prices” for the consumer

- **Stop guessing capacity** - “You can access as much or as little capacity as you need, and scale up and down as required with only a few minutes’ notice”

- **Increase speed and agility** - “The cost and time it takes to experiment and develop is significantly lower” because “new IT resources are only a click away”

- **Stop spending money running and maintaining data centers** - “Cloud computing lets you focus on your own customers, rather than on the heavy lifting of racking, stacking, and powering servers”

- **Go global in minutes** - “Easily deploy your application in multiple regions around the world with just a few clicks” which will also “provide lower latency and a better experience for your customers”

Cloud computing will continue to grow, so it is a natural progression to evolve OpenC2’s standard to support cloud commands. During our research this semester, our team focused on the two most popular cloud providers: Amazon Web Services (AWS) and Microsoft Azure. Within the cloud infrastructure services market, AWS leads with a 32% market share while Azure’s is second with a 21% market share (Mitzi, 2021). Combined, these two companies hold over half of the world’s cloud market, so we decided to exclusively research how both of these cloud services operate. By comparing the largest two cloud providers, we got a great sense of which services are commonly offered (e.g. storage, computing instances, etc.) and what information is needed/optional for equivalent commands. Ideally, a future team will research other cloud providers in addition to AWS and Azure to concretely abstract commands for the OpenC2 standard. This will be further explored in our “Further Work” section of our paper.

**Functional Areas and IAM**

In the beginning of the semester, we were asked to focus on one functional area of cloud service providers. A functional area is a collection of features, commands, and associated information that fits together to support certain enterprise needs. We ended up choosing Identity and Access Management, or IAM for short. IAM is a framework of business processes, policies and technologies that facilitates the management of electronic or digital identities. With an IAM framework in place, information technology managers can control user access to critical information within their organizations. Systems used for IAM include single sign-on systems, two-factor authentication, multi factor authentication, and privileged access management. These technologies also provide the ability to securely store identity and profile data as well as data governance functions to ensure that only data that is necessary and relevant is shared. As such,
we chose to focus on IAM because it is the foundation of other functional areas. Having proper control over user accounts and permissions—and the ability for them to securely sign into their accounts—is the basis for which many, if not all other cloud provider services rely on or utilize in some way.

The following table describes a few of the many other functional areas that exist, and gives some examples of AWS and Azure services that fall under each respective category:

<table>
<thead>
<tr>
<th>Functional Area</th>
<th>Description</th>
<th>AWS Services</th>
<th>Azure Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utility Compute</td>
<td>Assets to support general computation, such as virtual machines and containers. Includes configuring and managing compute resources.</td>
<td>EC2, Container services, Lambda</td>
<td>Virtual Machines, Container Instances, Kubernetes, Functions</td>
</tr>
<tr>
<td>Storage</td>
<td>Object and block storage, archive storage, backups</td>
<td>S3, Elastic Block Store, FSx, Glacier</td>
<td>Blobs, Archive Azure Files, Azure Disk</td>
</tr>
<tr>
<td>Database</td>
<td>Relational, column store, and graph databases; document repositories, digital ledgers</td>
<td>DynamoDB, RDS, DocumentDB, Keyspaces, Neptune, QLDB</td>
<td>Azure SQL, Cosmos DB, Table Storage, Conf. Ledger</td>
</tr>
<tr>
<td>Machine Learning, AI</td>
<td>ML model building analytic, and interference services; ML processing services</td>
<td>Comprehend, Forecast, Textract, Lookout, Polly, Rekognition</td>
<td>Cognitive Services, Azure ML, Text Analytics, Computer Vision</td>
</tr>
<tr>
<td>Cloud Management</td>
<td>Tools for monitoring and analyzing cloud usage, automating cloud activities, policy management</td>
<td>Cloudtrail, Cloudwatch, Trusted Advisor, Control Tower, Config</td>
<td>Resource Manager, Lighthouse, Sentinel, Policy, Automation, Advisor</td>
</tr>
<tr>
<td>Integration</td>
<td>Facilities to connect parts of a complex system, activate parts of a system, or manage communication across parts of a system</td>
<td>Simple Queue Service, AppFlow, EventBridge, SWF</td>
<td>Web PubSub, EventGrid, Service Bus, Logic Apps</td>
</tr>
</tbody>
</table>
Use Case

We brainstormed several use cases within the IAM functional area that we considered as important actions that we may want OpenC2 to be able to perform.

- An attacker has gained access to the identity system and has created several user identities to allow for future access to the system. We want to delete those user accounts.
- The user reports that they have lost their MFA device, and we want to deactivate MFA for that device so that an attacker can’t use it to get into their account.
- An attacker has gained access to an account, so we want to remove all privileges from the account.
- A vulnerability was discovered in some aspect of the system (ex: SQL injection vulnerability) and we want to make sure no one can exploit that vulnerability before it’s patched, so we temporarily take away users’ ability to make changes to the vulnerable area.
- There was a password breach, so now all of your users’ passwords are known. We want to change users’ passwords to a random string and email that new password to them so only the real owner can access the account.

The use case that we decided to pursue was deactivating a user’s MFA device. As individuals and businesses become more reliant on cloud services for computing, storage, and other applications, it is crucial to ensure that any data sent or stored on these cloud services is protected against unauthorized access. This involves Identity and Access Management (IAM), which ensures that intended users have the intended access to account resources. One method of preventing accounts from being compromised is through Multi Factor Authentication (MFA). MFA requires two or more pieces of proof of identity before access to the account is granted, such as a password and a code sent to the account owner’s phone number. This greatly lowers the chance of an account being compromised, since malicious hackers are less likely to have access to both pieces of identity information. Additionally, if a password breach occurs to the cloud service, MFA serves as an extra wall of protection.

If the MFA device is lost, it’s important to remove the connection between the account and the MFA device as soon as possible, because it poses a security hazard if an unknown person finds it and uses it to access the account. We would like to use OpenC2 commands to disable and/or delete the device, in order to re-secure the account.

OpenC2 Command Development

In order to develop OpenC2 commands, we initially brainstormed useful actions in the IAM functional area, such as creating users, creating groups, and other relevant actions. Gathering commands helped us better understand the broader scope of our use case’s functional area. Furthermore, we researched the implementation of each command in AWS and Azure to ensure they existed in both cloud providers. Through this research process, we discovered additional commands that we had not yet considered, including role assignment.
Next, we filtered out the commands to only include those that were relevant to our MFA use case. From this, we determined six useful commands: create a user, find a user, delete a user, change a password, create an MFA device, and delete an MFA device. Creating and deleting a user are necessary actions in the MFA use case, as users are mapped to devices. Additionally, finding a user is especially important for Azure, as commands often require a user id as an argument, rather than a username. With this command, a user’s id can be found by the administrator if the username is known. Moreover, changing a password is useful in the scenario that a person’s MFA device is compromised, as they can reset and secure their account password. Lastly, creating and deleting an MFA device is important in establishing a device or disabling it if it is compromised.

To develop OpenC2 requests and responses that fully incorporated elements in the corresponding AWS and Azure commands, we utilized the command line interface (CLI). By doing so, we were able to determine the similarities and differences between the both cloud providers' commands, which informed our OpenC2 commands. For creating a user, we found that AWS takes the username as the argument, while Azure takes the username, password, and user principal name. For deleting a user, finding a user, and changing a password, AWS takes the username, while Azure takes the user id. To account for this distinction between AWS and Azure in OpenC2, we ensured that the corresponding targets required either a username or a user id as an attribute. One interesting revelation discovered in this process was that Azure does not have CLI implementation for creating or deleting an MFA device, even though AWS does. Instead, this command can only be implemented in an API call (McLaughlin, 2021). The implications of this are discussed in “Further Work.” With a combination of this CLI research and presentation feedback, we were able to develop more informed OpenC2 commands and targets.

**MFA OpenC2 Commands**

Our new commands build on the existing features of OpenC2. We used existing actions, but we also describe four new targets:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>new_user</td>
<td>New-User</td>
<td>1</td>
<td>A user that doesn’t exist yet</td>
</tr>
<tr>
<td>31</td>
<td>existing_user</td>
<td>Existing-User</td>
<td>1</td>
<td>A current user</td>
</tr>
<tr>
<td>32</td>
<td>new_device</td>
<td>New-MFA-Device</td>
<td>1</td>
<td>A device that doesn’t exist yet</td>
</tr>
<tr>
<td>33</td>
<td>existing_device</td>
<td>Existing-MFA-Device</td>
<td>1</td>
<td>A current device</td>
</tr>
</tbody>
</table>

(addition to table 3.3.1.2 in the OpenC2 documentation (OASIS Open Command and Control Technical Committee, 2021))
Each of these target types is defined as follows:

**Type: New-User**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>username</td>
<td>String</td>
<td>1</td>
<td>The username of the new user</td>
</tr>
<tr>
<td>2</td>
<td>password</td>
<td>String</td>
<td>0..1</td>
<td>The password of the new user</td>
</tr>
</tbody>
</table>

**Type: Existing-User**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>username</td>
<td>String</td>
<td>0..1</td>
<td>The username of the existing user</td>
</tr>
<tr>
<td>2</td>
<td>user_id</td>
<td>String</td>
<td>0..1</td>
<td>The user ID of the existing user</td>
</tr>
</tbody>
</table>

*Usage Requirements:* Target MUST contain at least one property

**Type: New-MFA-Device**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>username</td>
<td>String</td>
<td>1</td>
<td>The username of the user the device will be associated with</td>
</tr>
<tr>
<td>2</td>
<td>phone</td>
<td>String</td>
<td>0..1</td>
<td>The phone number of the MFA device</td>
</tr>
</tbody>
</table>

**Type: Existing-MFA-Device**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>username</td>
<td>String</td>
<td>0..1</td>
<td>The username associated with the MFA device</td>
</tr>
<tr>
<td>2</td>
<td>user_id</td>
<td>String</td>
<td>0..1</td>
<td>The user ID associated with the MFA device</td>
</tr>
<tr>
<td>3</td>
<td>device_id</td>
<td>String</td>
<td>1</td>
<td>The device ID of the existing device</td>
</tr>
</tbody>
</table>

*Usage Requirements:* Target MUST contain exactly one of username and user_id

We created targets for users and MFA devices because there weren’t existing targets with the appropriate properties. Originally, we used only two targets: User and MFA-Device. However, we decided that targeting a user that doesn’t currently exist is fundamentally different from targeting one that already exists. Therefore, we split User and MFA devices into two targets.
The first action that we needed to implement was being able to create a user. This might need to be done in the aftermath of the use case we’re considering. The command uses the action “create” and the New-User target. The optional password argument is used for clouds like Azure, that don’t give the option of a randomly generated password. It returns basic information about the user once it’s created, including the user ID.

Create User
Command:
{
    “action”: “create”,
    “target”: {
        “new_user”: {
            “username”: “test_user”,
            “password”: “test_pass”
        }
    }
}

Response:
{
    “status”: 200,
    “results”: {
        “user_id”: “A1B2C3”,
        "accountEnabled": true,
        ...
    }
}

The next action we agreed on was a command to delete a user. In the case that a user’s account has been infiltrated, there might be cause to delete the account entirely. This command uses the action “delete” and the Existing-User target, which takes either a username or a user ID. This is because different clouds will use different fields to identify their users. For example, AWS identifies users by their usernames, and Azure identifies users by their user ID. The response is a status code indicating the success of the operation.

Delete User
Command:
{
    “action”: “delete”,
}
One way to protect a user’s account could be to change their password to something secure by auto-generating a new password. We wrote a command to do this with OpenC2. It uses the “set” action and Existing-User target. It returns a status code indicating whether or not the action was successful, and the new password. One possible implementation would be for the system to email this new one-time password to the user after receiving the response.

Change Password
Command:

```
{
  "action": "set",
  "target": {
    "existing_user": {
      "user_id": "1a2b3c"
    }
  }
}
```

Response:
```
{
  "status": 200
}
```

If an MFA device has been removed from a user’s account because it was lost, there should be a way to create a new MFA device for that user. This command uses the action “create” and the New-MFA-Device target. This target always takes a username or user ID,
because an MFA device has to be associated with a certain user, and has the option of including a phone number. This is because some clouds, such as Azure, use a call/text system for their MFA device, so you would need to provide the phone number. Other clouds, like AWS, use virtual MFA devices, usually an app on the user’s phone. The response returns the new device’s ID, and a Base32StringSeed if using a cloud like AWS that uses an app to authenticate. This string is entered into the app, and then that app can be used as an MFA device.

Create MFA Device
Command:
{
    “action” : “create”,
    “target” : {
        “new_device” : {
            “username”: “test_user”
            “phone”:”123-456-7890”,
        }
    }
}
Response:
{
    “status” : 200
    “results”: {
        “device_id”: “1234”,
        “Base32StringSeed”: “WCXXKPX…”
    }
}

The crux of our use case is removing an MFA device that has been lost or stolen. The command to do this has the action “delete” and the target Existing-MFA-Device. This target takes the device ID and the username/user ID of the account associated with the device. The response is a status code indicating the success of the operation.

Delete MFA Device
Command:
{
    “action” : “delete”,
    “target” : {
        “existing_device”: {
            “device_id” : “1234”,
        }
    }
}
Something that attackers sometimes do once they infiltrate a system is create several users with lots of permissions, in case they get locked out of the account they used to get in. We wanted to write a command to find suspicious users so they could be dealt with, as well as fetch important information about a user such as user ID and MFA device ID. This command uses the action “locate” and the target Existing-User, and can take a regex in place of the username in order to find users created automatically with similar names. The response lists the matching users as well as relevant information about them, including their user ID, when they last signed in, and the device ID of an MFA device associated with that account.

Find Users
Command:

```json
{
    "action": "locate",
    "target": {
        "existing_user": {
            "username": "test_user"
        }
    }
}
```

Response:

```json
{
    "status": 200,
    "results": {
        "username1": {
            "user_id": "12345",
            "last_signed_in": 303920 (DateTime),
            "device_id": "p2d34fg"
        },
        "username2": {
```
Further Work

One area of additional work would be to expand the choice of cloud providers that the commands are designed to support. For example, Google Cloud is another major cloud provider that would be advantageous to consider in a cloud-agnostic command system. Research would need to be done on how Google Cloud capabilities align with the OpenC2 cloud commands we developed, and what modifications would need to be made to the targets, arguments, and responses.

As previously mentioned, while we chose to develop commands relevant to the Identity and Access Management functional area of the cloud, there are several other functional areas to consider. Working on designing OpenC2 specifications for more functional areas would expand the capability of OpenC2 for the cloud. Doing work in this area would include identifying common commands in the functional area, recognizing similarities and differences between their uses in different cloud providers, and designing the OpenC2 commands and responses in a cloud-agnostic fashion. Additionally, since we only focused on one use case for IAM, other use cases could be considered in order to further flesh out OpenC2 commands relevant to IAM.

Furthermore, a potential next step is to develop a prototype implementation of a server and client relationship that can send, receive, and respond to OpenC2 cloud commands. A prototype would consist of several parts. First, the server component would be capable of constructing an OpenC2 command based on options specified by the user, and sending the command to a target client. The client program would be able to receive this command and carry out the action it specifies, and send a response back to the server. Each instance of the client program would be registered as either an AWS or Azure client. The specific client would be pre-configured with access keys for either an AWS or Azure account such that it would be able to perform the desired commands on those platforms, either through the CLI or API. Since the OpenC2 commands are cloud-agnostic, a client registered to either provider would be able to handle any command appropriately, performing the action on the respective cloud provider and sending a response back to the server.

One particular area of complexity in developing a prototype is the implementation of the command to create and delete an MFA device. As briefly mentioned, the AWS CLI includes commands to create and delete an MFA device, but Azure only has an API endpoint to do so. This highlights the need to be flexible in how the prototype makes calls to the cloud providers.
One solution would be to include functionality for executing both CLI commands and API calls within the prototype.

**Conclusion / Lessons Learned**

Some of the lessons we learned throughout this project were the significance of working directly in AWS and Azure, the similarities and differences between the two cloud providers, and the importance of OpenC2. We found that it was very important to work directly with AWS and Azure as it was vital to contextualizing our use case as well as understand the circumstances in which it would be applicable. As a result of working with both cloud providers, we learned about the many similarities and differences between them. A specific example of this would be how AWS takes usernames as the identifier whereas Azure takes an id as the identifier. Through OpenC2, these similarities and differences can be standardized so that a use case within all cloud providers can be implemented at once and utilized to create a secure cloud environment.

**Works Cited**


