UNSW
SYDNEY

COMP9447 Security Engineering Workshop

Automated Incident Response Project Report

Team 1

Jatin Gupta - z5240221

Judy Fan - z5157590

Jenny Kwon - z5222646

Tom Nguyen - z5122502

Mark Sonnenschein - z5025193

Bhumika Singhal - z5234799

Justin Mackintosh - z5160822

Table of Contents

Executive Summary

Are you a fast-growing start-up, large enterprise or a leading government agency who is trying to move their existing applications to the cloud and build nearly anything you can imagine?

Well worry not, we have got you covered with Amazon Web Services (AWS). AWS is the world's most comprehensive and broadly adopted cloud platform, used by many to lower costs, become more agile, and innovate faster. AWS has many services, and features – from infrastructure technologies like compute, storage, and databases–to emerging technologies, such as machine learning and artificial intelligence, data lakes and analytics, and Internet of Things. AWS is architected to be the most flexible and secure cloud computing environment available today. But the point to focus on is - Does everything work perfectly in real time?

Here we present you with a small application especially designed to demonstrate some of the possible threats that can be faced by any application. We have created a basic Medical CRM with two tables – Doctors and Patients. The tables store information about doctors and patients like name, age, phone number etc. The threats which we are demonstrating are divided into two categories – Internal and External.

In the coming sections, we have covered about the AWS facilities and framework along with the details of our application. We will also talk about the AWS facilities that we have used in our application and tried exploring them. We have given the details of the Incident Response Scenarios and their implementation. We implemented in total of 6 incident response scenarios which are combination of both internal and external threats.

The main purpose of the project is to have responses ready for some defined threats which can be applied to any application to make our lives easier. The document will end with a quick conclusion describing about our experience with working with the AWS.

Introduction

The importance of cybersecurity cannot be understated in today's technological world. With the recent proliferation in data breaches such as the 2018 Cambridge Analytica scandal, it has become even more important to develop a robust framework for cybersecurity.

Cybersecurity is a high priority in storing and managing healthcare records as this sort of information is extremely confidential and has the potential to be used in malicious ways such as for profiling by health insurance companies or blackmail. There have been various pieces of legislation for protecting health records such as The Privacy Act (1988) and HIPAA in the United States which contains provisions on the storage, use, and disclosure of these records.

Incident response is a method for handling security incidents and breaches. A good incident response plan allows an organisation to effectively identity and minimise the damage caused by a cyber-attack as well as make it easier to identify the root cause of such attack.

In considering incident response, we must consider both external and internal threats to an organisation. External threats deal with actors from outside the organisation gaining access to internal systems and data by breaching the external security barrier. Conversely, internal threats deal with already trusted actors such as employees and administrators who use their knowledge of internal systems and processes to damage and exfiltrate data. Many organisations overlook the threat that insiders pose, even though they are far more dangerous than external actors due to their presence inside of the security perimeter. By having good internal protections, we are also able to limit the damage that can be caused by external actors if they find their way into internal systems.

The AWS Well-Architected Framework

The Well-Architected Framework is a methodology that helps developers compare their architectures to the standards and best practices set out by AWS Solutions Architects. The idea behind the framework is that the AWS Solutions Architects have been building apps for many years and have run into most the design problems and considerations that are out there in the wild.

They have built a guide to help developers that don't have as much experience, to get familiar with some principles, that will avoid the need to reconsider the considerations that they have already polished. The framework is built upon the following five pillars: operational excellence, security, reliability, performance efficiency, and cost optimization.

Operational Excellence

Operational excellence is defined as: "The ability to run and monitor systems to deliver business value and to continually improve supporting processes and procedures."

Perform operations as code

This design principle informed our use of the AWS Cloud Development Kit (AWS CDK) to construct our infrastructure through code. This drastically reduced the occurrence of human error and meant when there were issues, we could just fix them and redeploy a fresh stack.

Annotate documentation

We didn't make a much use of the annotated documentation design principle, but certain actions implicitly leave a documentation trail such as the naming of our stack deployments. When we used the GitHub API to redeploy our frontend, we could also insert information about the redeploy within the request body which would be annotated in the rebuild of the frontend.

Make frequent, small, reversible changes

This design principle was also informed by our use of the CDK. It meant that people could work on their own stacks independently which reduced workflow blockages, helping us to increase the frequency of deployments. We tried to keep each deployment small so that we could make sure each change we were making had the desired effect on the infrastructure. This was aided by the fact that we were relatively new to AWS and CDK, meaning that our trial and error strategy was implicitly constructed of small and frequent changes. The nice thing about the CDK is that we could use Git to version control our infrastructure meaning that when mistakes were made, reverting to a safe and stable state was trivial.

Refine operations procedures frequently

We excelled at refining our operations considering how new we were for much of this project. We met regularly and changed our ways frequently in favour of more efficient, less error-prone

or easier methods. One particularly notable example is when we migrated to Typescript from Python as the language that we used for the CDK. This is due to some issues we ran into with the documentation available for the CDK API in Python. Another example is the fact that we learned how to abstract our infrastructure deployments by using CloudFormation, but further refined our workflow to use the even more abstract CDK quite early in the project.

Anticipate failure

Fortunately, this was the easiest design principle to adhere to as this is the entire premise of the project. We have scripts that will attempt to make unsolicited changes to certain parts of our infrastructure. Our industry-leading incidence response mechanisms have been crafted to deal with every stage of IR: preparation, identification, containment, eradication, recovery and lessons learned.

Learn from all operational failures

One notable method through which users of our incidence response solutions can learn from operational failures is from our intelligent logging of the incidents themselves. We use Amazon Simple Notification Service to send alerts of detected failures to the team responsible for the maintenance of the demo application.

Security

Security is defined as: “The ability to protect information, systems, and assets while delivering business value through risk assessments and mitigation strategies.”

Implement a strong identity foundation

We ensured that each serverless component only had permissions over the specific resources which they needed. We kept all permissions in a centralised privilege management solution, specifically AWS Identity and Access Management (AWS IAM).

Enable traceability

This was very implicitly tied to the core of our project, and we required extensive logging capabilities in order to facilitate much of our IR. We took this further by notifying the administrators when an alert requires their attention.

Apply security at all layers

Defence in depth was very well demonstrated through our use of the AWS Web Application Firewall (AWS WAF) as an external hardening layer and very specific permissions on the intertwined internal layers of serverless components. Furthermore, our use of serverless itself gave us the added layer of security over every layer of the stack below the business layer of logic.

Automate security best practices

Our extensive use of the CDK means that our infrastructure is largely represented as code which makes it easier to modify the security configuration of the infrastructure and automatically redeploy it. As security best practices change, it's easy to add them to the CDK code and redeploy the entire stack transactionally. Of course, with the infrastructure being code we can automate the version back to older one, in the case where, this might be needed by reverting to an older git commit.

Protect data in transit and at rest

We used Cognito for our authentication system which is responsible for issuing authorisation tokens to the authenticated users. These tokens are then used to fetch restricted information from the backend such as doctor details and patient records. We also use HTTPS to encrypt all web traffic.

Keep people away from data

We didn't do so much of this, because the application itself is not very developed as we spent our time on the IR itself. In a more developed application, this would be seen using forms to input data (i.e. full frontend CRUD capabilities on all database entities).

Prepare for security events

Once again, this was essentially a by-product of our assignment. We were constantly running threat events against our application and successfully defending against the threats as shown in the demonstration presentation.

Reliability

Reliability is defined as: "The ability of a system to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand, and mitigate disruptions such as misconfigurations or transient network issues"

Test recovery procedures

Since our main motive was to come up with IR, testing our issue recovery was an implicit part of the project. We tried testing our AWS resources against all the possible threats that we could come up with in each attack scenario. The recovery strategies were strengthened through a healthy mix of successful and failed simulations.

Automatically recover from failure

We made extensive use of log analysis to ensure that very specific key performance indicators would trigger remediation when certain behaviours were detected, and thresholds exceeded. This allowed us to automatically recover to a safe working state almost immediately after an unsolicited behaviour occurred.

Scale horizontally to increase aggregate system availability

The great thing about cloud platforms is that they are designed with scale in mind. Whether vertical scaling, horizontal scaling, or a mix thereof is required, well architected systems can often be trivial to accomplish. Furthermore, the serverless strategy that we employed made scaling an absolute no-brainer. For example, Lambdas are completely standalone pieces of stateless and idempotent code. If your requests exceed the capability of one running instance of a Lambda, an abstracted load balancer will automatically spin up a new Lambda instance and load balance between the available instances. This means a generous subset of failing instances is rarely noticed by the users of the system as there isn't a single point of compute failure.

Stop guessing capacity

The infrastructure behind serverless components is actively managed by the platform rather than the developers meaning that when there is no use of the system, resources can be completely scaled down to zero and when there is a larger-than-expected load on the system, extra instances can be automatically spun up. This means that rather than having to prophesise the maximum capacity of a system and always maintain that largest capacity, systems can take advantage of cheaper operating costs when little or no system resources are needed and if there is a spike in resource requirements without notice, rather than resource starvation leading to outages or reduced availability, systems will still be able to operate.

Manage change in automation

Once again, CDK helped us achieve automatic updates to the infrastructure. This means that most of our work revolved around amending the CDK and there were very few overheads when it came to making amendments to the infrastructure itself.

Performance Efficiency

Performance efficiency is defined as: "The ability to use computing resources efficiently to meet system requirements, and to maintain that efficiency as demand changes and technologies evolve."

Democratize advanced technologies

One of the best examples of the democratisation of advanced technology is that we made good use of AWS Cognito. Conventionally many people fall into the trap of thinking that auth isn't so hard to do, however the more you know about auth, the more you know that it really is something you should leave to the experts. For this reason, we used Cognito to handle the entirety of our auth requirements.

Go global in minutes

We didn't make much use of the multi-region benefit that AWS offers their customers. We just deployed all our work in the ap-southeast-2 region to get the lowest latency possible.

Use serverless architectures

The entirety of both our example corporate application (Doqutor) and our complete incidence response stack was built completely using serverless components. This removed a huge amount of the operational overheads we would have otherwise faced. It also gave many of us exposure to a new generation of infrastructure so that we could use the it for our future projects.

Experiment more often

We each used different stacks with allowed us to experiment with changes to the infrastructure and code without negatively impacting each other's workflows. Certain things we could have better experimented with such as our choice of database, however, often this was due to the time constraints of the project. We were, however, constantly experimenting with serverless due to the extremely unfamiliar nature of this methodology to most of us.

Mechanical sympathy

A good example of mechanical sympathy is how we used Dynamo DB for containing the doctor and patient records but used an S3 bucket for serving our website. These are both storage technologies, however, we concluded that for our project our data storage was best split appropriately over these two different services.

Cost Optimization

Cost optimization is defined as: “The ability to run systems to deliver business value at the lowest price point.”

Adopt a consumption model

Serverless is generally billed using a consumption model. With our entire application and IR infrastructure being constructed completely from serverless components, we very much adhered to this design principle.

Measure overall efficiency

Due to the small scale of our application, there wasn’t much we could have actioned when it came to be improving our efficiency. Whether you have 100 requests each minute or 1, you’re still using, for example, a single instance of a Lambda. We did however monitor our usage so that we could in the future use that to get useful insights into our efficiency.

Stop spending money on data centre operations

We are proud to report that 100% of (the $0) expenditure was completely using AWS’s infrastructure. We very much adhered to this design principle.

Analyse and attribute expenditure

We did find it very easy to estimate our costs, as we could easily see how many instances of each serverless component we would need. The major cloud providers make it easy to convert that into a final price with transparent pricing and even pricing calculators.

Use managed and application level services to reduce cost of ownership

We made appropriate use of managed services such as Cognito and SNS to take care of application higher level, well-defined tasks. This helped remove the operational burden of having to implement some of these applications ourselves.

Our Solution

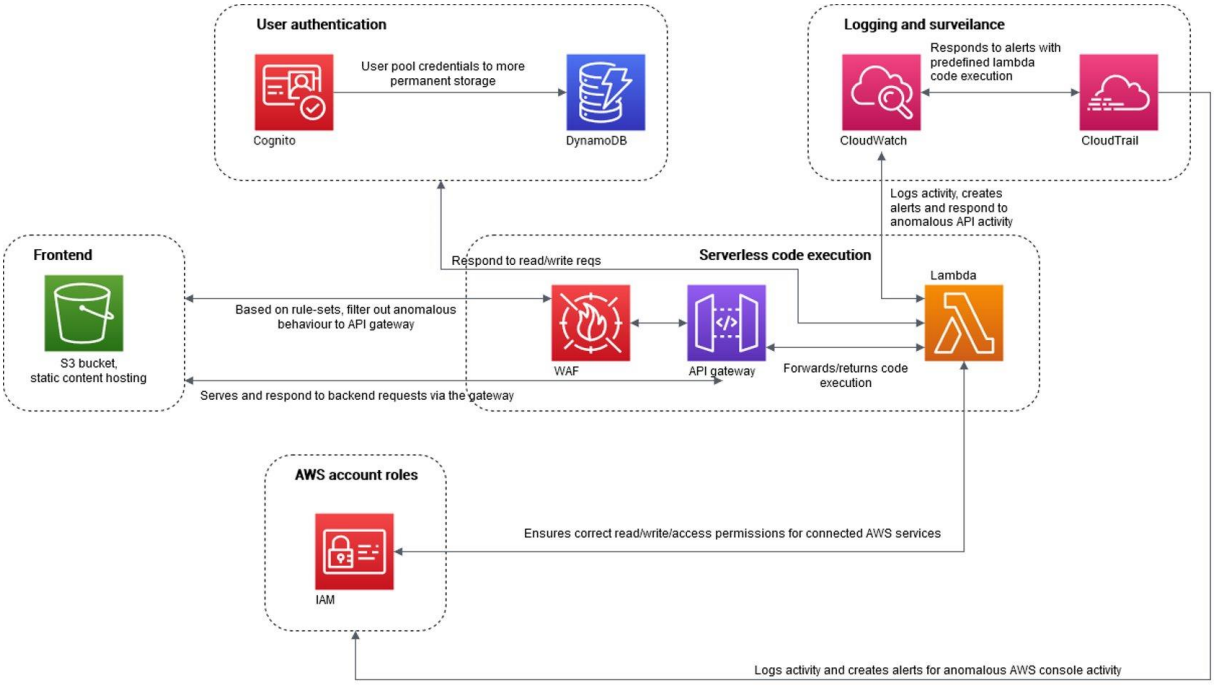

FIGURE 1: The Medical CRM Architecture Diagram

Though AWS provides us with plethora of great services to work with. But for this project we wanted to be keep it lean and utilize the bare minimum for this project. We have created a Medical CRM with create, read, update and delete functionality on two entities - Doctors and Patients.

For the front end, it is built with Vue as the JavaScript framework, Vuetify as the component library and Nuxt as the server-side rendering framework. It uses the exposed HTTPS endpoints from AWS Lambdas functions for the API calls and uses GitHub Actions for CI/CD deployments to AWS S3 buckets. We are not using server-side rendering for the current use case.

For the backend, we are utilizing a complete serverless stack. We are using DynamoDB as our database of choice. We decided to go with a document-based database because changing any field in the entity would be far easier than creating SQL scripts to update the schema in SQL based databases. We are using AWS Lambda functions as the CRUD API's and tied them to an API gateway which directs the request to the correct endpoint. We have also utilized an AWS WAF (Web Application Firewall) in front of the API gateway to protect it against some common web exploits that may affect availability, compromise security or consume excessive resources. We are using AWS Cognito for user authentication and authorization. Cognito talks to the frontend and provides JWT. For logging, we have utilized CloudWatch and CloudTrail. CloudWatch stores the logs for all the services used in AWS and CloudTrail saves all the events by AWS users. IAM is being used to block or to attach policies to the AWS users.

Our business logic is very simple for the application. We have simulated a health CRM where we have two entities, doctors and patients. It should be noted that this part of the application exists to employ and demonstrate our IRs and does not represent a complete or ideal CRM system.

Databases and entities

Patients can log in to the website and view information about doctors through the API. Doctors, while logged in, can view and edit doctor and patient records through the API. Cognito manages the database of website users, where each have a type doctor or patient. The databases used by the API are additional DynamoDB databases – one for patients and one for doctors. These databases are updated automatically on creation of new users in Cognito.

API

The API is structured as represented below using HTTP request notation, with the structure repeated for the /patients resource.
The API architecture uses a REST approach, with the URL uniquely specifying resource through the format 'collection/item', and the HTTP request method/verb specifying the operation. This system is clear, confirms to HTTP semantics and allows all our functions to be included within the same API endpoint. This made setup and maintenance of both API Gateway and the frontend API interaction easier. Results are returned as JSON as defined in the models in the API schema (api-schema.ts).

```
GET /doctors: list all doctors
POST /doctors: Create new doctor
GET /doctors/{id}: Get information on doctor with given id
PUT /doctors/{id}: Update information on doctor with given id
DELETE /doctors/{id}: Delete doctor with given id
```

The doctor's and patient's APIs differ in the precise fields making up a record, and in the information returned by a list request – listing doctors returns all information on all doctors,

while listing patients returns only basic information, and further information must be requested with a specific get request. This distinction was made because we are more concerned about the privacy of patient information, and get requests allow more logging of which records are accessed and by whom. The doctor information stored in the web-accessible database is intended to be publicly available, while patient information is intended to be accessed only by specific parties.

API authorization

Authorization for API calls is handled by Cognito. Each request must be accompanied by a valid (issued within the last hour and not revoked), signed JWT provided by Cognito to a user upon sign in. Cognito's user database includes patients and doctors, with a type attribute to differentiate them. For our purposes, Cognito/website users are created within AWS by administrators. Additional authorization based on user type, and whether a token has been revoked, is implemented within CRUD lambdas by calls to a common library. This does not use a custom authorizer, for reasons discussed in the credentials and honeytoken sections.

Our Experience with AWS Tooling

Overall, these tools were extremely intuitive especially CDK and CloudFormation as they helped us to deploy and teardown our stack numerous times. It also allows us to version our stack on GitHub, so that any person can deploy and rollback the stack in order to test things.

CDK

An initial problem that we faced with the CDK was that it was hard for multiple people to be working on a single stack at the same time as one person's changes often overwrote the changes of someone else. To fix this, we wrote a small wrapper on top of the CDK to dynamically generate a stack name for each person. This allowed us to iterate faster as we can all develop our stacks independently and merge at the end.

Another issue was that the CDK was still fairly new, so there were a few bugs that we had to work through. A particular issue that was annoying was that deploying Cognito via the CDK would seem to match on the surface, but then it threw some weird errors when we created users and logged in. We found that by logging into the console and unchecking/rechecking the checkboxes on the client settings page would fix the "an unexpected error occurred" problem, but we were not sure why. A similar issue with CDK Cognito lead to all userpools being deployed with an SMS role (https://github.com/aws/aws-cdk/issues/6943). At one point we thought that this was the cause of the "unexpected error occurred" problem, but since it was not and it seemed beyond our ability to fix, in the end we just ignored it. We also reverted to using CloudFormation for the frontend stack, as the documentation was much more mature. CDK was definitely speedier as we were able to rely on VSCode to do our type checking, so there was much less trial, and error compared to the CloudFormation route.

Due to file permissions, deploying lambdas through CDK sometimes proved problematic. The local file permissions on lambda code (read/write/execute for owner, other etc.) are maintained when lambdas are packaged and uploaded using CDK. This means that these files must be readable by 'other' users on our computers, which is inconvenient and a security hazard in its own right. Additionally, for one of us, this permission uploading didn't work properly, and we found that files with identical permissions and positions in the folder hierarchy did not deploy with the same permissions. This resulted in lambdas failing execution because files were not accessible by the Lambda executor. To work around this, we locally moved lambdas into whatever folders they worked in as necessary.

We were unable to deploy a subscription filter on a lambda log group using CDK. When the log group of a lambda function object is referenced, a custom log retention resource is created. However, this log retention resource failed creation because (according to CloudWatch) a singleton lambda within the CDK files could not execute due to the file not being accessible. CDK deploy froze until it timed out and reverted the deploy with the message that a custom resource could not stabilise. We used a python script using boto3 to build the subscription filter instead, gathering the source log group name from the console, AWS CLI or Boto3.

Other AWS tools and services such as Lambda and CloudWatch/CloudTrail was straightforward, and the tooling was excellent. The detailed and powerful logging capabilities of CloudWatch really helped us to debug our Lambda functions, and API gateway's authorization capabilities allowed us to totally rely on managed services for user management and authentication.

Incident Response Scenarios & Implementation

Case 1: External/Internal, Detecting Unsolicited Frontend Modifications: Attempted Vandalism

Threat

The frontend for our example application is build using Vue, Vuetify and Nuxt. When a new version of the web application is pushed to GitHub, GitHub Actions will run a CI/CD pipeline which will generate static HTML, CSS and JavaScript files and then deploy them to an S3 bucket. Please note that pushes to the “dev”, “preprod” and “prod” branches on GitHub will inform the pipeline which respective AWS project to deploy the new frontend build to. A potential attack vector we noticed was that a bad actor could create or modify files in the S3 bucket. This could be from misconfigured authorization settings on the S3 bucket, or even an insider with S3 permissions that wishes to tamper with the bucket. We have decided that the only changes to the frontend considered legitimate must come from an execution of the entire CI/CD pipeline (in an ideal world the CI/CD pipeline will have many rounds of tests, and any modifications that bypass these safeguards could wreak havoc on the end-product).

Detection

To mitigate this potential attack vector, we have set up a watcher on the CloudTrail logs which detects changes to the S3 bucket. Whenever there is any modification in the S3 bucket, it gets logged in the CloudTrail. This is detected by the CloudWatch which triggers a lambda function to take the perform the required tasks.

Response

If the changes were enacted by the CI/CD pipeline (i.e. the " frontend-deployment" user with its own set of public and private keys), then the changes are ignored. If any other user makes the changes, then a lambda is triggered which returns the CI/CD pipeline to create a fresh copy and deployment of the frontend to replace any of the unsolicited changes to the S3 bucket. Furthermore, a notification is also sent to the dedicated body to notify about this change. This is done with the help of AWS SNS.

Implementation Issues and Design Choices

While implementing this, we found that the lambda function would be called on every single modification to the bucket, so we had to limit the concurrency to the lambda function to 1, so that there won’t be 50 lambdas running at once on a deploy. Additionally, we added a cooldown timer of 15 minutes to the lambda so that only the first change within the time period would trigger the rebuild and notify the administrator. This was good enough as when the administrator gets this email, they would hopefully be checking the status for at least longer than 15 minutes.

We also used an external platform to host our CI pipeline as we were more familiar with how it works. To interface with the CI, we used GitHub's REST API to trigger the rebuild. This meant that we created an additional IAM user and stored the secrets within GitHub. Therefore, the security of our frontend also depended on the security of GitHub's platform. Of course, we would have easily rolled back the bucket by using S3 versions, but we also figured that if the attacker already had access to our bucket, they could just delete the old versions to make the rollback fail. In a real-world situation, this would normally be prevented by setting up bucket policies so that only our deployment IAM user would have write access to the bucket. However, an improvement that could be made would be to use CodeBuild and CodeCommit as to not rely on external services.

Demo

For the purpose of demo, we have created a demo file by the name "demo.txt" and it is uploaded on our S3 bucket. Now a lambda gets fired and a fresh copy of frontend is deployed replacing the changes that took place in the S3 bucket. The frontend is now back to its original form and a notification has been sent to the subscribed channel.

Case 2: Internal, Disabling of CloudTrail Logs: Compromised AWS Account

Threat

CloudTrail is a very important service from AWS. This service enables AWS account to maintain and store logs throughout the AWS account. These logs can be analysed for governance, compliance, operational auditing, and risk auditing of your AWS account. It separates the logs into different event types which can be filtered and queried when required. It maintains action history logs of all actions performed through AWS Management Console, AWS SDKs, command line tools, and other AWS services. These logs are saved in S3 buckets with industry standard names and folder structure which can help user to analyse and troubleshoot problems. With support from CloudWatch, we can add rules on special cases and fire events based on CloudTrail event. There can be 5 CloudTrail instances for each AWS account.

Since it is very important for an AWS account to have logs, stopping the logs would prove to be a security risk for the AWS account. If the AWS CloudTrail is switched off, any user can update any setting and we would not have logs to have them accountable for it. Hence it is essential to keep CloudTrail running. Further, disabling logs is a common move for an attacker after infiltrating a system, so monitoring CloudTrail serves as an effective indicator of compromise.

Detection

When a trail is turned off by a user, we flag that user as malicious. It could have been the case that the access to the AWS account has been compromised of some user. For our use case, we

detect the when the CloudTrail has been turned off. CloudTrail emits an event called, “StoppedLogging” we use this event to trigger response for the event.

Response

We respond the event by switching on the CloudTrail instance and blocking the user by attaching the policy of “deny-all”. We also emit important log information about events through simple notification service to the subscribed channels. For blocking the user, we use the lambda functions. Lambda function utilizes serverless code which can execute at an event and we do not need to run them an event detection process on a server. We use an instance of IAM from python AWS SDK, boto3 for attaching the policy to the user. We also send notifications to inform the subscribed channels about “StoppedLogging” and “StartLogging”. These notifications also contain the username of the arguable malicious AWS account user.

Implementation issues & design choices

We faced a challenge while working with TypeScript Lambda as they were not giving proper error messages and hence were difficult to debug. Hence, we shifted to Python as the language of choice for lambdas.

Demo

We have created a demo script to set up an IAM user. As we will block a user during the simulation of the IR, hence we have created a script to get user credentials in a json file and the system user can add new user’s configuration keys to the local system to act as the demo user. We have also created a command line wizard which can call all CloudTrail instances and switch off the first one. It can also call get-trails command to check if the demo user is blocked or not.

Case 3: External, Data Exfiltration Detection via Honey Tokens: Data Exfiltration

Threat

Data exfiltration is described as unauthorized copying, or retrieval of sensitive data from a computer system. Organizations with high-value data are particularly at high risk of these types of attacks, whether they are from outside attackers or trusted insiders. Data exfiltration is among organizations’ top concerns today. Data breaches can be very damaging to an organisation’s reputation, share price and profitability, and socially focused organisations will also be concerned about the personal impact on their clients. Many of security professionals said they have experienced data breach at their current work or organization.

Detection

Thus, we came up with detecting access to honey tokens as an indicator of compromised or abused credentials, and of data exfiltration. Our implementation of a honey token, or honey record, is an item added to a database that does not represent any real-world entity. A genuine

user (i.e. a doctor, receptionist) would only access patient records of patients they are working with, so patient records that are not assigned to any doctor should not be accessed. If they are accessed through the API, we take it to signal that a doctor's website account, or their current authorization token, has been compromised. Perhaps an attacker is attempting to download certain attractive patient records, or many random patient records.

Response

When this indicator is triggered, our response is to sign out and disable the website user, and then invalidate their current authorisation token. We also notify the administrators.

Implementation - The IR implementation has three components:

1. Python script to generate honeyrecords and subscription filter
2. An event filter on the lambda log groups that triggers a lambda function
3. The lambda function that executes response

Honeyrecords generation**:** We have a python script (boto3) to:

a) generate honeyrecords
b) add them to the database
c) create a subscription filter on the CloudWatch log groups of certain lambdas (by default, patients_get and patients_delete) which perform API CRUD operations. The script can also be used to remove a previously added subscription filter and its honeyrecords.

Subscription filter: If a honeyrecord is accessed, the subscription filter will pick up the database item id from the lambda's logs and trigger the blockuser lambda with information about the invoker.

User blocking: The user blocking lambda will

a) Sign out and disable the Cognito user.
b) Add the authorization token to a DynamoDB database of revoked tokens until it expires. Cognito does not natively support token revoking before expiry. Before authentication for each API call completes, this database is checked for the provided token. DynamoDB deletes expired tokens automatically (and at no additional cost) using TTL.
c) Notify administrators of the event through Simple Notification Service. Information sent includes the time of the honeytoken access, the username and IP of the responsible user, and the success or failure of the blocking attempt. An administrator can then review the API use, contact the offending user and respond further with actions such as requiring a password change.

Implementation Issues and Design Choices

The main issue with implementing this was (what seems to be) a bug with CDK, which causes deployments of stacks that refer to the log group name of a lambda object to fail – a custom

logRetention resource is created, but this resource creation freezes until CDK stops the deploy attempt. To work around this, we created the python script to create the subscription filter. Once this script was in place, we used it to dynamically, upon request, generate and destroy honeyrecords and corresponding subscription filters.
We were also limited by the fact that a subscription filter is set as a string of maximum length 1024, which sets a hard limit on the number of honeyrecords that can be monitored using this approach. Each log group can only have one subscription filter.

A significant design choice made with this IR was whether to use a Cognito custom authorizer for the checking of the revoked tokens database, or to call a library function from each of the API lambdas to perform the check before allowing the database access. As discussed in case 2 (gaining access to credentials), a custom authorizer has benefits of decoupling authentication from the lambdas, reducing code repetition, cleaner CDK stacks, and allowing expansion of authentication techniques. However, a custom authorizer also adds significantly more complexity and code than is needed for this task and adds the delay of another lambda, thus increasing the latency of our API. We opted to do the database check in the lambdas. Using a custom authorizer is a possible future improvement.

Flaw: if a password is compromised, an attacker could sign in repeatedly and get several tokens. Then they could use them one by one - when one is blocked, they can use another, as other tokens will remain valid. From this perspective it would be better to enforce token invalidation based not on a database of invalidated tokens, but on the user being disabled or signed out. However, this limits flexibility as the system could not be used to block a particular stolen token while allowing the genuine user to continue using the service. Perhaps an ideal implementation would be a combination of both – one system intended for one-off stolen tokens, and another intended for stolen accounts.

Demo

A honeyrecord generation script, given the CloudFormation stackname, adds (somewhat) randomly generated honeyrecords to the patient's database and creates a subscription filter as explained above. The script uses a backing script which takes more specific inputs (table name, source log group names, destination ARN).
A simulation script, given the API endpoint and a Cognito authorization token (retrieval of which is assisted by a third script), requests a list of patients and then makes calls to the patients_get lambda requesting more information on each patient. When a honeyrecord is requested, the response is triggered and future requests (after a short delay) will return that the incoming token has been revoked.

Case 4: Internal, Modifying Database Outside Application: Non-Patient Reading/Writing Patient Data

Threat

In any application, one of the most important tasks is to maintain the security of the database. In this Medical CRM, we have created two tables – Doctors and Patients. In any medical facility, the administrators can be given the access of the Doctors information but is it secure to give them access of the Patient's data like their prescriptions, medical history or in our case their insurance_id.

Addressing Correct Read Permissions: To avoid such a situation, we have created a customer managed IAM policy known as "BlockPatientTable" which is applied to a group called "adminusers" who have "AdministratorAccess" policy attached to it. The policy is an entity which when applied to a resource or identity defines their permission. This policy is used to deny the complete access of the patient table by any member of the mentioned group. Our main objective here is to be ready for a situation where a user despite having the BlockPatientTable permission tries to access the table from outside the application.

Detection

In order to track the changes committed on the table, AWS provides us with DynamoDB Streams. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table. You can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items.

Response

We have enabled a stream on our patients table which triggers a lambda function "cloudtrail_ddb_access" whenever there is a change in the table. The stream emits an event which is used by the lambda to check if the event is "Modify". It compares the old value of the insurance id with the new one. If there are any changes, the lambda stores the old values of all fields and deletes the current entry from the table. The lambda then generates a new entry with the restored values. Since this event is not a Modify event, the lambda will not be triggered and hence the changes made by the user will not be reflected in the table. Along with this we also send a notification using AWS Simple Notification Service to the subscribed channel indicating that there has been a change made in the table. The notification contains the primary key, which is called id, on which the changes has been made.

Implementation Issues and Design Choices

We started our IR with the idea that if a person is trying to access to the patient's data, the user will get aware of the fact he/she does not have the permissions to do. If the user gets the entry to the AWS Console, he/she can go the IAM and remove the attached policies. To avoid such a situation, we decided to attach a customer managed IAM policy which will block the user from

using the IAM console. For this task, we were required to have the username from the logs so that we can apply the policy. After further research, we got to know that the DynamoDB logs are not stored in the CloudWatch and hence we cannot trigger a lambda on a CloudWatch event.

Then we came across DynamoDB Stream which captures any modification done on a table when enabled. After exploring it further, it came to our notice that DynamoDB Stream event does not have any field which gives us the username of the account from which the modification took place. At this point, we had to drop the idea of attaching the IAM block policy to the user.

We finalized our IR to a situation where if the faulty policy is applied to the user for denying the access to patient's table and a user tries to exploit it by modifying some records, the updated record will be deleted and a new record with the old values will be generated. We will also notify the responsible authority by sending an SNS.

Demo

The IR is split into three pieces of functionality:

1. Python incident demonstration wizard
2. DynamoDB Stream detecting all modifications and consequently fires a lambda function
3. Lambda function that uses the Stream's old and new images to create a transaction "rollback"

The demo requires the user to create a new IAM user that has "AdministratorAccess" policy attached, however it must not be in the "adminusers" group in order to have read functionality. The Python wizard conducts the following steps:

1. Declare the Patient table name and id of the record to be demoed
2. Attempt to call GetItem on declared id
3. Alter insurance_id to something other than the one stored
4. Views the altered record
5. Views the same record after the rollback has occurred

Case 5: External, Gaining access to Credentials: Compromised CRM Account/Brute-force Attack

Threat

This is routinely featured on the OWASP top-10 list as Broken Authentication. By breaking authentication, an adversary can compromise user accounts, as well as user data if an administrator's credentials are found.

Detection

Our solution uses a Cognito user pool to handle all aspects of user management, from signing up new users, to multi factor authentication and password management. We used Cognito's built-in

features to perform the detection for us, mainly because it is already implemented but also because we didn't have access to the internals API to retrieve metrics from Cognito.

Response

When the user tries a password too many times, Cognito will start to rate limit their attempts and will also gradually increase the time which they can retry the password. Additionally, if a user logs in from a new location, Cognito will also trigger a Multi Factor Authentication event, which will require the user to enter in a code from a text message. On top of this, we have configured a lambda function to send an email to the user on any suspicious login attempts.

Implementation Issues and Design Choices

A challenge that we faced when implementing this use case was that Cognito did not give us too much information via its APIs, and that we couldn't configure things more granularly such as the maximum number of password attempts. It was also quite hard to change the details of users after creating them, as most of these fields had to be changed via the Cognito API due to the Cognito management UI being limited.

Additionally, creating the Cognito user pool using the CDK had some weird errors which left us scratching our heads for days. As mentioned earlier, sometimes the CDK deploys produce unexpected results as some extra parameters have been added in by default and are also hidden in the user interface, so we couldn't really tell what was going wrong. However, we did eventually find a fix to this, which was to explicitly define all the options and parameters.

We could have created a user interface to make it easier to create users, but we felt that it was not necessary for the purposes of the demonstration. So instead, we opted to use scripts that use the Cognito API to create both doctor and patient accounts. Note that these accounts are different from AWS accounts, and do not have access to the AWS cloud in any way.

A possible improvement to this scenario would be to use a Lambda to add custom logic to process our login request, such as to bar the user or just fail their login. However, this would add some extra complexity as we would also be reimplementing some of Cognito's features.

Demo

For our demo, we used a Selenium script to simulate an attacker trying to guess random passwords in order to log into a system. The script works by generating a random password and then submitting it to Cognito. We also used a script to create accounts in the Cognito pool as we didn't have an interface to do it from.

Case 6: External, Rate limiting and WAF: API Attack

Threat

Rate limiting is important to web applications as it prevents resources from being exhausted, as well as detecting and preventing data exfiltration. In the cloud where resources are easily auto scaled, another risk known as economic denial of sustainability can surface where a malicious attacker attempts to inflate the bill of a cloud customer by triggering the autoscaling capabilities within the customer's apps. Other risks involve cross-site scripting and injection attacks, which could compromise the security and data of an application.

Detection

A Web Application Firewall (WAF) can be used to detect and prevent most of these attacks. With Amazon's WAF, we set up a rule to limit requests to 128 every 5 minutes, which is plenty for a user at home to view and update their medical records. We also enabled other options such as XSS and an IP reputation list, to better secure our application against other attacks that we may have not thought about.

Response

Once the WAF detects more than our threshold of requests in a certain timeframe, it immediately starts to block access to the web application. In addition to this, a sample of the requests coming through the WAF is taken for further analysis for the developers to gain some insight from possible attacks. If the number of blocked requests from a source reaches a further 128 requests from the baseline of blocked requests, we will then notify the administrator about the problem.

Implementation Issues and Design Choices

We opted to use the default blocking time and didn't add any extra functions to handle the blocking as we believe that adding extra scenarios would introduce complexity to our application and would also lock out legitimate users. For example, creating a lambda function to block a set of IP addresses for an hour would mean that people legitimately sharing IP addresses such as CGNAT, schools and companies would be falsely blocked due to a single bad actor.

Again, we did hit on a few CDK errors, as the WAF was not fully built into CDK at the time of this project, so we had to resort to using a CfnStack. There were a few quirks with how the CDK handled the CloudFormation stack, as some of the parameters had different names. But overall, we were able to get what we were looking for in the end.

However, the WAF was a bit limited as it did not allow us to set different rate limits based on particular IP addresses. This would have allowed us to allow significantly more requests from medical clinics, as the doctors and receptionists would be using the system much more heavily than an average user logging in from home to check and update their records. For that reason, the

request limit in a real-world deployment would have to be higher to cope with the demands of the clinic users.

Future improvements to this scenario would be to use a more robust messaging platform such as Slack or text messaging, as emails can be overlooked. More aggressive blocking could also be an option, due to the sensitivity of the data contained within.

Demo

For the demo, we used a URL script to simulate someone maliciously hitting our endpoints with useless requests. From doing the demo, we see that the WAF is a little bit slow to react as it was able to let through about 300 requests before it finally started blocking the API requests. We would know when the requests started to be blocked as that was when the status changed over to 403 instead of 200. Additionally, we also demoed the CloudWatch action as when the alarm is triggered, an email would get sent to our mailbox via SNS.

Conclusion

Participating in this course was definitely an exciting and rewarding experience, as it enabled us to learn and understand the roles that cybersecurity plays in industry. Also experimenting with AWS's serverless stack and plethora of offerings was both challenging and fun.

By being a part of this project, we found that the AWS serverless stack was very mature and made implementing the incident response relatively easy compared with rolling everything yourself on EC2 virtual machines. We really liked the integration that AWS provides between the different services which keeps the process of building a cloud-native solution quick and simple. Some improvements that we would like to see would be in the CDK, as it is a very promising architecture but currently is a bit buggy.

While coordination in COVID lockdown was something we got better at over time, we feel that the depth and quality of our learning under normal circumstances was more difficult to replicate. While we're proud of knowledge we gained, our learning about the AWS platform and incident response would have been even greater if situation was different, with many more hours of face-to-face contact with Nick and Pierre. We would have benefited from more guidance in taking full advantage of the breadth of AWS services, and in the types of incidents we should prioritise responding to. Current iterations of UNSW security courses do not discuss about incident response in any depth, and we've come to the conclusion that it is a difficult topic to teach yourself.

With that said, we are all very proud of the learning and progress achieved through the term as a team, and we enjoyed being exposed to the potential of AWS.

References

Amazon Web Services. (n.d.). *Well Architectured Framework.* Retrieved from Amazon Web Services: https://d1.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

STRIDE (Security). (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/STRIDE_(security)

The Privacy Act 1988. (n.d.). Retrieved 5 4, 2020, from Office of the Australian Information Commissioner: https://www.oaic.gov.au/privacy/the-privacy-act/