**Contextual Security Dashboard**

**Final Project Report**

**Final Requirements, Design,
Implementation & Testing**

**Interactive Intelligence**

**CSC 492 Team 3:**

**Veronica Alban**

**Joseph Kuttickal**

**Blake Lassiter**

**Wes Toler**

**North Carolina State University**

**Department of Computer Science**

**December 12, 2016**

**Executive Summary**

Effectively monitoring all the security data needed for a given platform is an essential, though sometimes difficult task. This difficulty is exacerbated when dealing with a highly decentralized system. This challenge has presented itself in Interactive Intelligence’s PureCloud platform. PureCloud is built using a microservices architectural design pattern. This essentially means that it consists of several autonomous services working together by exposing their APIs over a network. A centralized location to store and view security data would benefit PureCloud’s security team.

Given the need for a unifying system, Interactive Intelligence asked our senior design team to create a contextual security dashboard that can display security data for the PureCloud platform collected from back-end modules. This dashboard has the ability to display security data as well as the ability to filter out the less important data so that security teams may focus on the more pressing security concerns. This security dashboard is, most importantly, usable for all microservices created in the PureCloud platform. Since Interactive Intelligence is constantly creating new microservices, the goal is to seamlessly integrate them into the dashboard to allow visible notifications in a quick and usable manner. Our goal was to create a centralized location to access all types of reports and view notifications for each one. The system filters and sorts data that has been retrieved through a RESTful API.

Our team has created a single page dashboard that displays the security data for one module. We also completed a black box testing suite to ensure that our system complies with the functional requirements. Additionally, we created unit tests for our RESTful API using Django.

**Project Description**

**Sponsor Background**

This senior design project was sponsored by Interactive Intelligence, a software company that provides cloud services for customer engagement, communication and collaboration, designed to help businesses increase productivity and reduce costs. The company was founded by Dr. Donald Brown in 1994. Their main product is the PureCloud Platform. The PureCloud Platform was built using a microservice architecture on top of Amazon Web Services. Each microservice operates independently so individual teams can develop loosely-coupled software that allows for greater freedom. PureCloud is a versatile platform that can be used to spur teamwork on Collaborate or connect to your customers with Engage, a customer engagement platform. The company was acquired by Genesys for $1.4 billion in September of 2016.

Jarrod Sexton

Senior Security Engineer

4307 Emperor Blvd #300, Durham, NC 27703

703-593-7529

Jarrod.Sexton@inin.com

**Problem Statement**

Within the PureCloud platform, there are several scripts that generate security reports containing information about different potential vulnerabilities. These scripts are referred to as modules. Currently, the security reports of modules are not centralized to one location, which makes it challenging to view the most recent or critical alerts. Within Interactive Intelligence's platform, there are only a few modules that we need to collect information for, but each can have many reports to be displayed.

**Project Goal and Benefits**

The goal of this project is to create a dashboard that collects critical security notifications from backend modules on the PureCloud platform and displays them to the end users. The dashboard also needs to be flexible enough that it can be configured to support already existing modules on the PureCloud platform, and allow new modules to post their data.

Interactive Intelligence’s security team will benefit from the security dashboard as it centralizes all the important security data on a human readable interface. Security personnel can use the information displayed to quickly identify possible problems and vulnerabilities. On a larger scale, it will benefit the company and its customers because the security team will be able to react faster to security issues as they arise. Therefore Interactive Intelligence will be able to increase the company’s profits and customers’ satisfaction.

**Development Methodology**

In order to maintain a steady and efficient workflow we have decided to follow an agile software development methodology. We have divided the project into smaller manageable pieces, making them easier to test and debug along the way. Much of the testing requires dependencies to be setup for the entire project which will require more work on the implementation and design before testing can be fully completed. Much of the testing required Standup to be held regularly, at least twice a week, in order to make sure the team is making progress in the right direction. Within our project build, we have three iterations, which are outlined in the Task Plan as Alpha, Beta, and Gamma releases. Each iteration lasted approximately 3 weeks. Each week we held a 30-minute meeting with Jarrod Sexton, who is our sponsor with Interactive Intelligence. Meetings were held using the PureCloud website, which allows us to either video chat, voice chat, or send messages to each member of the group.

**Challenges**

One of the main challenges we faced during this project was picking the technologies that we were going to use to build our program. The original project gave us a lot of flexibility when choosing the implementation methodology. We initially thought that by having so much flexibility the project would be much easier but we found out quickly that this was not the case. The design itself has changed drastically from how it was originally planned. It was not until near the end when we finally settled on one solid design.

An additional challenge was designing an API that could accept multiple different categories of data, all with different sizes, fields, and nesting structures. We solved this by choosing JSON as the single format by which the API would accept data; this helped because JSON is very flexible in terms of the way data is structured. It's also easily parsed through and can be stored in a database as is.

Originally, our design dealt with a static webpage designed using Jekyll which made it hard to connect our back-end REST API to the website. As we continued struggling and exploring other technologies that were available we decided on using AngularJS which solved two major problems we had. The first benefit of using Angular was a smooth end to end connection between our database and website. The other benefit was the ease of adding dynamic functionalities to the front-end.

As a final requirement, the application needed to be deployed onto Amazon Web Services (AWS). AWS is a cloud platform which allows for the database, API, and webpage to continually run in a safe, remote, environment. AWS proved to be much more challenging to deploy than expected. The first block that we encountered was a lack of permissions to create, or use, services within AWS. These permissions were granted to us by our sponsor, but it took more time to explore AWS. The initialization of a relational database, and EC2 instance for the API and website took a considerable amount of time and effort. The application is not currently deployed to AWS due to configuration issues of PostgreSQL being of version 9.3 on AWS, but our project requiring 9.6. Attempts to modify the AWS configuration were unsuccessful.

**Resources Needed**

In order to successfully implement our dashboard, we'll need a number of software tools to build our application. These include Django, AngularJS, Python 3.0, PostgreSQL and Bootstrap. All of these items are freely available for download. An EC2 instance and an RDS instance will be required to host the application and database. Both EC2 and RDS are Amazon Web Services, and as such, have a cost associated with them. Our sponsor has provided us with a sandbox environment, giving us permissions to create these instances as needed.

**Requirements**

The following are functional requirements for the security dashboard. Users shall be able to create, select and view all current modules as well as add new ones when needed. The system will also be responsible for notifying the user of any new security alerts that were updated. The system shall also have multiple filtering options to allow a user to narrow down data. The date is one of the filtering options which allows the user to search for security data in incrementing or decrementing date order as well as the ability to filter by a range of dates or single date. Another filtering option is module name which will allow the user to type in the specific name of the module needed. Lastly a sorting filter shall be provided so the user is able to search for modules in alphabetical order.

1. **Functional Requirements**

1.1 Ability for user to select each module to view data

1.2 User able to add new modules

1.3 Filter security data

1.3.1 Filter by date range

1.3.2 Filter by module name

1.3.3 Sort modules alphabetically

1.3.4 Sort modules by ascending/descending notification date

1.4 User shall not be permitted to delete modules through the website

1.5 Notifications of high priority security entries shall be sent to PureCloud Webhooks

The following are our non-functional requirements for the security dashboard. We need a system with fast response time. We also want to be able to have seemingly instantaneous updates of database tables whenever valid information is received by the system. Along with this, the system must run continuously and provide fast feedback to users on requested modules. Lastly the database must be expandable since every new module will create tables in the database.

1. **Non-Functional Requirements**

2.1 Must have seemingly instant response time when buttons are clicked

2.2 Must update database as soon as valid data is sent to the UI

2.3 Must be able to store large amounts of data

2.4 Must provide current module data to the user within a few seconds

The following are constraints that our security dashboard must follow. In order to integrate the dashboard into our sponsor’s company we must make sure that the dashboard can run on Amazon Web Services. The system also must integrate with PureCloud's Webhooks since this is how Interactive Intelligence gets its security notifications. Lastly the system must be decoupled from the rest of the PureCloud platform meaning that must to run as an independent microservice.

**3. Constraints**

3.1 The system shall run on AWS

3.2 The system must be integrated with PureCloud's Webhooks

3.3 The system must be decoupled from the rest of the PureCloud platform

**High-Level Design**

Based on the requirements of our project, we have developed a design that is suitable for fast processing time, effective integration with software written in multiple languages, and will represent the data from PureCloud that is essential to the Interactive Intelligence security team. Figure 1 can be referenced for a visual aid to the description of our design and data flow.



**Figure 1: Design of Data Flow**

Our system’s front and back end will be running on an Amazon Elastic Computing Cloud (EC2) instance. EC2 is Amazon Web Services’ cloud computing platform. The operating system running on this EC2 instance will be Amazon Linux. Our backend development will use Django, a Python-based web development framework. The Django application is used to implement the REST API for interacting with our database of security reports. It provides, among other things, the ability for users to post reports to the database, as well as access the reports already residing there. For this project, we are using the PostgreSQL database. SQL databases are good to use with Django projects because they allow you to utilize Django's object relational mapping feature. PostgreSQL is particularly good to use with Django because it has packages that you can import when you're creating your models to give you access to types you wouldn't otherwise have. One such example is the JSONField, which is not natively supported by Django's Model class, but incredibly useful if part of your JSON input isn't predefined.

For our front end we used Bootstrap, which is a front end web development framework that contains components for HTML, CSS, and JavaScript, making stylish and responsive user interfaces easier to create. Bootstrap allows for quicker development of our front end interface, since there so many capabilities already included. We also used Angular to handle the databinding for our front end. Additionally, Angular has ways to handle interacting with our backend database using the API calls we developed. Angular also provides a way to parse through the data returned by our API. It has directives to iterate through the list of key/value pairs that makes up JSON formatted data.

**Low-Level Design**

The data flow starts with an external module making a POST call using our REST API to submit JSON formatted data to our dashboard's backend Django application. The JSON data sent must follow a specific format. Firstly, it must be sent as a list of items that our serializer can loop through, even if it's just a list containing one item. Each item must contain the fields: module, title, severity, and data. The "module" field is a string field describing which module on PureCloud the item comes from. The "title" field is also a string field, and it contains whatever title the originator of the item decided to give it. The "severity" field is an integer field that indicates the level of importance of a particular item. The range is from 1 to 3, 1 being the most important or severe. The "data" field is a JSON field that accounts for the variance between items coming from different modules. An example of this formatting can be seen in Figure 2.

The Django app will then use a serializer to parse the JSON data and store it in an object known as a model. The model is what Django uses for object relational mapping. When data is stored in a model object, and that object is saved, it will be added to the database in a table that was automatically created by Django based on the fields of the model object.

When the frontend displays data, it makes a GET call to the Django app. At this point, that data is now in model format, so it needs to be serialized back into JSON format and sent to the front end as a response. The front end can then parse that data and display it to the user.



**Figure 2: Example of JSON formatted data from the Amazon Linux AMI Security module**

In Figure 3, a list of all reports for one type of module is displayed below the metadata of the module. When a module is loaded onto a new tab it is helpful to display the information that distinguishes that module, as well as the list of reports for it. Figure 3 also shows much of our general front end design. We have a sidebar listing the modules for which there are reports in the database. When the user clicks one of the sidebar items, they will see a screen similar to the one shown in Figure 3. On the top right, we have a "Notices" button that will display all the most important/severe data.



**Figure 3: Web UI view of one module with all reports**

**Implementation**

Our project is broken up into three iterations. The first iteration handled the dashboard displaying reports only having to do with the Amazon Linux AMI Security advisories. This is one of many types of reports our dashboard will be expected to support by the end, but is a good starting point. To complete this iteration involves having a functional REST API that can accept POST calls from external modules, as well as accept GET calls from our frontend in order to retrieve data from our backend. The frontend view for this iteration will be very basic, as our main concern is that it is able to retrieve and display data correctly.

Our second iteration will involve more work on the frontend. We want to implement a home page, so that instead of the user being greeted with an empty screen when they open a new tab, they are given a screen that gives them some options for what to view. An example of a potential option would be a button for viewing the latest notifications on high priority security reports. Apart from changes just to our frontend, we want to make sure that our dashboard is capable of handling reports other than just for advisories. An example of another type of report we might ensure our dashboard can handle is one containing data on security groups in Amazon Web Services that are overly permissive, or have an SSH that is "open to the world".

Our third iteration will focus on giving the user tools to filter and sort data. Options for filtering include date range and priority. Our front end will allow the user to modify the view of data that is displayed. Additionally, we moved our project to Amazon Web Services. This involves setting up an EC2 instance to put our Django and Angular projects. We'll also need to set up an RDS instance to host our PostgreSQL database server.

**Test Plan**

In the development process of our project, we plan to follow best practices of agile development, which will keep a focused and organized schedule of short term deliverables and roles for all team members. There will be several tools used to test various layers of our project, since there is both a front-end and back-end requirement.

Our back-end development was outlined in the design section above, which demonstrates how we plan to process data from the Interactive Intelligence server with an API call that will collect data into a PostgreSQL database. The first thing to test is that the API correctly obtains the information for each module. Django automated testing can be used to test that our API calls are formatted correctly. We will also use Postman, which provides a visual interface that can be used to run API calls and return the results for quick access in testing live data.

A localhost connection can be made to run our application. We will use Selenium web testing to perform automated tests to ensure that a user can step through the web pages and correctly enter prompts. Along with the automated Selenium tests we have created a set of blackbox tests that will be used for manual testing of the front end. We can also connect to PureCloud's Webhooks and receive live data from our sandbox account chatroom. This helps to ensure that we have the data transmitted from a request to the database, then to both PureCloud and our application, completing the data flow just as in it would be live environment.

**Black Box Test Plan**

In the following section, several user level tests are listed in a black box test plan. Each test has a set of instructions for the user to follow that will allow them to run each test. The expected result of each test is also listed.

**Test setup**

* + Run Postgres server
	+ Run the Django Server
	+ Run Angular website
	+ Open Postman for API calls
	+ Using Postman, Post in the following files
	+ alas.json
	+ sensitive\_s3\_buckets.json
	+ insecure\_security\_groups.json

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID  | Description  | Expected Results  | Actual Results  |
| ViewModule  | 1. Click on the first module listed in the sidebar
 | 1. The content area to the right should change to display security data entries that deal with that module
 | 1. The content area to the right changes to display security data entries that deal with that module
 |
| WriteNotes  | 1. Click on the first module in the sidebar
2. Click on the title of the first module report shown in the right content area.
3. In the text box below the module information write "Testing comments"
4. Press the submit button
5. Refresh the page
 | 1. Text area should say "Testing comments"
 | 1. FAIL not implemented yet
 |
| MarkAsResolved  | * 1. Click on the first module in the sidebar
	2. Click on the title of the first module report shown in the right content area.
	3. Click on the dropdown that says status. And select Resolved
	4. Press the submit button
	5. Refresh the page
 | 1. Resolved is selected
 | 1. FAIL not implemented yet
 |
| MarkAsViewed | * 1. Click on the first module in the sidebar
	2. Click on the title of the first module report shown in the right content area.
	3. Click on the dropdown that says status. And select Viewed
	4. Press the submit button
	5. Refresh the page
 | 1. Should show status as viewed
 | 1. FAIL not implemented yet
 |
| SortByPriority  | 1. Click on the Severity header column
2. Click on the header again
 | * 1. Reports should be in ascending order by severity
	2. Reports should be in descending order by severity
 | * + 1. Reports are in ascending order by severity
		2. Reports are in descending order by severity
 |
| SortModuleByStatus | 1. Click on the status header column
2. Click the header again
 | 1. Reports should be in order stating resolved
2. Reports should be in order stating with viewed as first
 | * + 1. Reports are in order stating resolved
		2. Reports are in order stating with viewed as first
 |
| SortByDate  | 1. Click on the Date header
2. Click on the Date header again
 | 1. Reports should be displayed in ascending order by date
2. Reports should be displayed in descending order by date
 | 1. The reports of the module are sorted in ascending order by date
2. The reports of the module are sorted in descending order by date
3.
 |
| SortByModules  | 1. Click on the module column header
2. Click on the header again
 | * 1. The column should be sorted by module name in ascending order
	2. The column should be in descending order
 | * 1. The column is sorted by module name in ascending order
	2. The column is in descending order
 |
| ViewNotifications  | 1. Click the tab for notifications
 | * 1. The tab selected should now be Notice
	2. The content area should only display reports with severity of 1
 | * 1. The tab selected should now be Notice
	2. The content area should only display reports with severity of
 |
| GoToHomepage  | 1. Click on the first module
2. Click on the home tab at the top right of the screen
 | * 1. The tab selected should be Home
	2. The content area should shoe all the modules that exist
 | 1. Correct tab is selected
2. All modules are visible
 |
| sortByTitle | 1. Click on the Title header
2. Click on the Title header again
 | * 1. Title should be in ascending order
	2. Title should be in descending order
 | * 1. Title should be in ascending order
	2. Title should be in descending order
 |

**Results**

**Selenium Testing:**

For the front end we used automated testing with selenium to make sure that the flow of the website was correct. During the selenium testing we automated the black box tests as well as added more detailed test cases to make sure that the website was running as we expected. The majority of the tests passed when we did a full run of the test cases. The tests that were not passing involved the website’s ability to update certain information in the database. The website is supposed to be able to update comments and status for each individual report that exists. Because we have yet to implement these features, all the tests cases involving comments and status updates failed.

**Django Testing:**

Django testing allows for mock data to be stored to a database, which is emptied each time the test is run. The test data allows us to verify that we can reference data purely through the API, rather than a view of PostgreSQL database or through a website. Tests have been completed such that the functionality required for GET, DELETE, PATCH, and POST calls are satisfied. At this point, there is no testing for authentication, since it was not implemented in our project via Django.

Coverage of a Django application can be determined via the "coverage.py" script, however, for our project it proved to be inaccurate and verbose. The coverage reported checked statement coverage of our Django files, but statement coverage was not a valid determination of what had been accomplished.

**Task Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Owner(s) | Due Dates | Status |
| Status Presentation | Wes Toler | 9-7-16 | Complete |
| Alpha Release:* Finalize user interface design
* Design template for JSON files
* Module import (API calls and API development)
* Database storage of modules
* Ability to pull messages from the database for each module
 | ALLVeronicaJoeBlakeWes/VeronicaWes/Veronica | 10-2-16 | Complete |
| Design/Implementation Document – Alpha | ALL | 10-5-16 | Complete |
| Meet with sponsor for Alpha feedback | Blake | 10-5-16 | Complete |
| Beta Release:* Implement visual interface of dashboard
* Expand on functionality in the back-end
* Selenium/Postman testing
 | ALLVeronica/JoeBlake/WesALL | 10-23-16 | Complete |
| Design/Implementation Document – Beta | ALL | 10-24-16 | Complete |
| Meet with sponsor for Beta feedback | Veronica/Joe | 10-26-16 | Complete |
| OPR3 | Veronica/Joe | 10-31-16 | Completed |
| Gamma Release:* Final bug fixes
* Implement search/filter options for modules back-end
* Implement search/ filter options for modules front-end
* Finish implementing buttons on the front-end
* Integrate back-end with Webhooks
* Final testing
 | ALLALLBlake/WesVeronica/JoeVeronica/JoeBlake/WesALL | 12-5-16 | Completed |
| Design/Implementation Document – Gamma | Veronica | 11-21-16 | Completed  |
| Posters and Pies | Wes | 12-7-16 | Completed |
| Installation documentation  | ALL | 12-13-16 | Completed |
| Final Presentation | Wes | 12-13-16 | Completed |

**Suggestions for Future Teams**

As of now, our dashboard exists as a single page application. This implementation does not include URL routing. If we had actual URL routing, we could have views for individual reports correspond to unique urls that you could send to someone as a hyperlink for easy access. For us to implement this ourselves we would have had to do refactoring of our Angular application on a scale that was infeasible given the amount of time we had left in the project when we realized this problem. Implementing this in the future would be good as it would enhance our dashboard's webhooks integration capabilities.

One extension you could make to our dashboard is to have actual analytics for the data that is coming in. For example, you could have some sort of line chart that shows the number of high severity reports coming into the database over time. You could also have a separate line on this chart for each module, so that you can see what areas of your system are encountering the most vulnerabilities. Something like this would make a good homepage for the dashboard.

Another extension of our dashboard includes using webhooks. Though we didn't use this functionality, you can apparently use a webhooks call to create a new chatroom in PureCloud Collaborate. An idea would be to have a way to create a new chatroom for each report in the dashboard. The way this could work is, on the report detail page, you could have a button that says "Create Discussion" or something to that effect. This would then create a PureCloud chat room for that rep, so that people can discuss it without using our comments section in the app.

A final potential addition to our dashboard would be to support actual users. Right now, the dashboard doesn't have separate states for different users, and adding this could allow different people using the dashboard to personalize their experience with it. They could have their own "watchlist", where they flag certain reports that they want to closely follow the progress of. Implementing this would involve some sort of authentication system, like OAuth or SAML. We recommend SAML as that is what Interactive Intelligence uses currently for authentication purposes.

**Authorship**

|  |  |  |
| --- | --- | --- |
| Section | Primary Author | Secondary Authors |
| Executive Summary | Wes Toler | Blake Lassiter, Veronica Alban |
| Project Description | Blake Lassiter | Wes Toler |
| Resources Needed | Wes Toler | Blake Lassiter |
| Requirements | Veronica Alban |  |
| Design | Wes Toler | Blake Lassiter |
| Implementation | Wes Toler | Joe Kuttickal |
| Test Plan & Results | Veronica Alban | Blake Lassiter |
| Task Plan | Veronica Alban |  |

**Team Contact Information**

Veronica Alban, Lead Coder & Planning Manager, valban@ncsu.edu, 336-497-1771

Joe Kuttickal, Editor-In-Chief & Quality Manager, jgkuttic@ncsu.edu 919-428-5488

Blake Lassiter, Team Leader & Lead Tester, balassit@ncsu.edu, 336-906-1736

Wes Toler, Lead Architect & Development Manager, mwtoler@ncsu.edu 336-782-8589