

Team-level slides

Group 2

An Automated Web Vulnerability Scanner

Problem: Web Application vulnerabilities are the most common form of security vulnerability, composing 25% of all breaches. They are difficult to detect as they require specialized training to understand and most web applications have wide surface areas.

Solution: Create an automated web vulnerability scanner that will catch simple vulnerabilities to reduce the attack surface area of web applications.

Key Features: The application, at its simplest level, would function through being supplied a domain/url to a website. Additionally, subdomains and other endpoints may be specified. Our scanner would then execute a number of simple enumeration procedures commonly seen in pentesting (e.g. directory busting) and display the data to the user (e.g. Commonly named directories). The user can also supply additional information to be able to run more specialized scans (e.g. specify where user data may be reflected in the client to perform a Cross Site Scripting scan).

Why This Project: Web application vulnerabilities, particularly client side ones, often affect not only the web application's provider but also the users. Many smaller web application providers also do not have the resources to contract audits of their applications. Because of this, the potential damage from and attack surface for web application vulnerabilities is extremely high. An automated scanner would hopefully be able to remediate at least the vulnerabilities that automated malicious vulnerability scanners check for.

Tagged Repository: <https://github.com/MrLarryMan/Tensile/tree/milestone-5-submission>

Milestone/Issues: <https://github.com/MrLarryMan/Tensile/milestone/4>

The Builders

Anay Gandhi - *Scrum Master & Project Manager*

- UI Issue: Job Specification Page

Hoa La - *QA Lead & Mediator*

- UI Issue: Login page

Larry Liu - *Cybersecurity Architect & Timekeeper*

- UI Issue: Saved Job Page

Mason Lemberger - *Systems Architect & Notetaker*

- UI Issue: Analytics and History Page

Historical Timeline

[illegible]

Individual Team Member Slides

-Hoa La-

Assigned Work Summary

- Work done this milestone: wrote 5 API endpoints (GET, GET, POST, PUT, DELETE) related to /saved-jobs API; integrated backend apis with previous frontend work
- + PR (merged) link: <https://github.com/MrLarryMan/Tensile/pull/34>
- Status: this milestone is interesting as I get to work on real backend apis and integrate with previous frontend code→ Completed & closed all issues in this milestone
- Links to all issues you are assigned in the milestone that were not closed: none

Feature Demonstration:

Features implemented:

1. [GET] get all saved jobs
 2. [GET] get a single saved job by jobId
 3. [POST] create new saved job
 4. [PUT] update saved job
 5. [DELETE] delete saved job
- Progress: all are completed and integrated well with frontend js code:
<https://github.com/MrLarryMan/Tensile/tree/hoa-milestone6>

Code Structure & Organization

- Code structure: follows template structure of separating frontend and backend folders; in backend, split into models, routes, controllers, and middleware.
- Critical components: routes, models, controllers, and middleware components are organized separately in their folders.

```

  ✓ backend
    ✓ controllers
      JS savedJobController.js
    > middleware
    ✓ models
      JS SavedJob.js
    ✓ routes
      JS savedJobRoutes.js
    ✓ src
      JS server.js
    .gitignore
    {} package-lock.json
    {} package.json
    > data
  ✓ frontend
    > imgs
    # analytics.css
    <> analytics.html
    JS analytics.js
    JS fetch.js
    <> index.html
    <> job_specification.html
    JS job_specification.js
    # login-styles.css
    <> login.html
    JS login.js
    <> register.html
    JS register.js
    <> saved_job.html
    JS saved_jobs.js
    JS script.js
    # styles.css
```


Front-End Implementation

- Structure: components, styles, and utilities are split into different html, css, js files – clean and nice
- Backend integration: by fetching/ calling the right API endpoints exposed in the backend code
- Challenge: map to the right backend API → Solution: use console.log for debugging on both sides and iterate till success

```
document.addEventListener("DOMContentLoaded", function () {
  const jobSelect = document.getElementById("job-select");
  const detailBox = document.getElementById("detail-box");
  const deleteModal = document.getElementById("confirm-delete-modal");
  const jobFormModal = document.getElementById("job-form-modal");
  const jobForm = document.getElementById("job-form");

  let currentJobs = [];
  let selectedJobId = null;

  const API_BASE_URL = 'http://localhost:3000/api/saved-jobs';

  initPage();

  jobSelect.addEventListener("change", handleJobSelect);
  document.getElementById("run-btn").addEventListener("click", handleRunJob);
  document.getElementById("edit-btn").addEventListener("click", handleEditJob);
  document.getElementById("delete-btn").addEventListener("click", handleDeleteJob);
  document.getElementById("confirm-modal-yes").addEventListener("click", confirmDelete);
  document.getElementById("confirm-modal-no").addEventListener("click", () => deleteModal.close());
  document.getElementById("cancel-form").addEventListener("click", () => jobFormModal.close());
  jobForm.addEventListener("submit", handleSubmit);

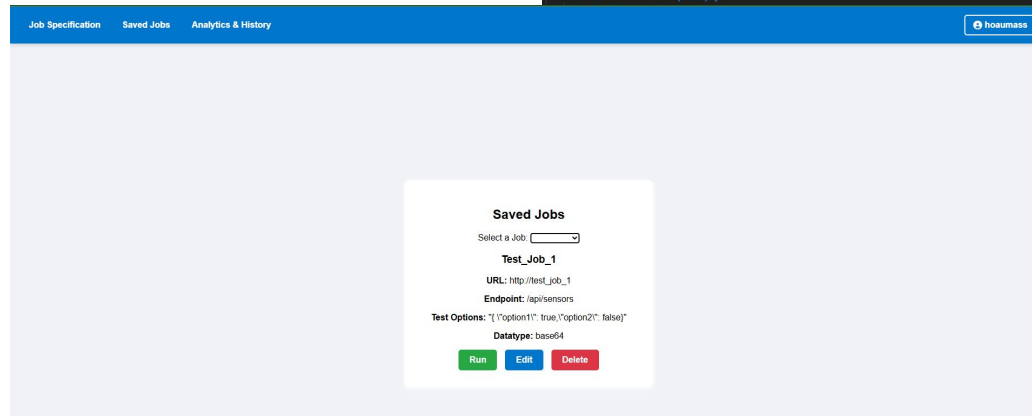
  async function initPage() {
    try {
      await fetchJobs();
    } catch (error) {
      detailBox.style.display = "none";
      console.error("Error initializing page:", error);
      alert("Failed to load saved jobs. Please try again.");
    }
  }

  async function fetchJobs() {
    try {
      const response = await fetch(API_BASE_URL);
      if (!response.ok) throw new Error("Failed to fetch jobs");
      currentJobs = await response.json();
      populateJobSelect(currentJobs);
    } catch (error) {
      console.error("Error fetching jobs:", error);
      throw error;
    }
  }

  function populateJobSelect(jobs) {
    jobSelect.innerHTML = '<option value="">-- Select a Job ---/options';
    jobs.forEach(job => {
      const option = document.createElement('option');
      option.value = job.job_id;
      option.textContent = `Job ${job.job_id} || Job ${job.url}`;
      jobSelect.appendChild(option);
    });
  }

  function handleJobSelect(event) {

```



Back-End Implementation

- Structure: models, routes, controllers, and middleware components are split into respective folders – ease for collaboration
- Frontend integration: by exposing clearly the right API endpoints, along with clear documentation how to use
- Challenge: deciding and parsing correct data type for request body → Solution: review Express.js request and body types to use the right data type. Make this clear via docs.

```
// GET: get all saved jobs
exports.getSavedJobs = async (req, res, next) => {
  try {
    const jobs = SavedJob.getAll();
    res.json(jobs);
  } catch (err) {
    next(err);
  }
};

// GET: get single saved job -- by jobId
exports.getSavedJob = async (req, res, next) => {
  try {
    const job = SavedJob.getById(req.params.jobId);
    if (!job) {
      throw new NotFoundError('Job not found');
    }
    res.json(job);
  } catch (err) {
    next(err);
  }
};

// POST: create new saved job
exports.createSavedJob = async (req, res, next) => {
  try {
    const newJob = SavedJob.create(req.body);
    res.status(201).json(newJob);
  } catch (err) {
    next(err);
  }
};

// PUT: update saved job
exports.updateSavedJob = async (req, res, next) => {
  try {
    const updatedJob = SavedJob.update(req.params.jobId, req.body);
    if (!updatedJob) {
      throw new NotFoundError('Job not found');
    }
    res.json(updatedJob);
  } catch (err) {
    next(err);
  }
};
```

js Specifications Saved Jobs API Endpoints & History

houmann

Saved Jobs

Select a Job

Test_Job_1

URL: http://test_job_1

Endpoint: /api/sensors

Test Options: '{"option1": true, "option2": false}'

Datatype: base64

RunEditDelete

Challenges & Insights

- Obstacles faced and lessons acquired: learnt how to integrate full code from frontend to backend. Navigating and connecting frontend, backend, and memory database.
- Key takeaways from working within a collaborative team environment: communicate well to avoid overlap and work towards a common goal for the team

Future Improvements & Next Steps

- Future improvements or features: real working (cloud) database integration for live serving
- Technical debt or aspects that could be optimized further: none for now.
- Issue link: <https://github.com/MrLarryMan/Tensile/issues/35>

Individual Team Member Slides

-Anay Gandhi-

Assigned Work Summary

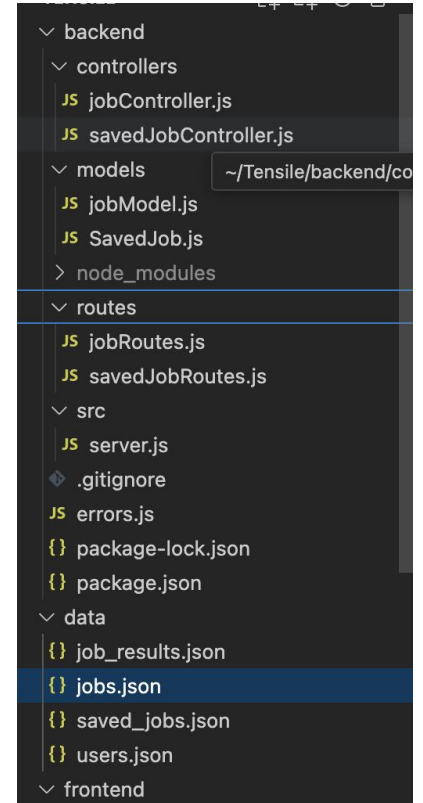
- + Work done this milestone: Wrote 5 API endpoints (GET, GET, POST, PUT, DELETE) related to the /api/jobs API for the job specification page; integrated backend APIs with the job specification frontend to enable full CRUD operations and persistent JSON storage
- + PR (merged) link: <https://github.com/MrLarryMan/Tensile/pull/40>
- Status: this milestone just established the backend of the job spec page, which I have completed. → Completed & closed all issues in this milestone
- Links to all issues you are assigned in the milestone that were not closed: no remaining open issues for Milestone #6

Feature Demonstration

- [GET] get all jobs
- [GET] get a single job by jobId
- [POST] create new job
- [PUT] update job
- [DELETE] delete job
- <https://github.com/MrLarryMan/Tensile/tree/job-spec-backend>

Code Structure & Organization

- Code structure: clear separation dividing frontend and backend. Backend is divided into dedicated folders for each of the different parts – models, routes, controllers, src
- Critical components: as previously mentioned, models, routes, controllers, src all have their independent attributes



Front-End

- All UI logic for job spec page is within dedicated JS file, with clear separation from HTML/CSS
- Interacts with backend by sending fetch requests to appropriate /api/jobs endpoints for saving/entering job data, ensuring seamless data flow
- Challenge: ensuring frontend matched backend data expectations, and so that the API calls reached the intended endpoints
 - Log statements and console debugging to manually track it

Job Specification

URL:

Endpoint:

Endpoint Data Type:

JSON

Select Tests:

☐ Test 1

☐ Test 2

☐ Test 3

☐ Test 4

```
26         return;  
27     }  
28     if (!isValidEndpoint(endpoint)) {  
29         showMessage("Please enter a valid endpoint (e.g., /v1/products). Endpoints must st  
30         return;  
31     }  
32  
33     showMessage("Ready to run tests (not implemented).");  
34     console.log("Running tests with the following details:");  
35     console.log("URL:", url);  
36     console.log("Endpoint:", endpoint);  
37     console.log("Data Type:", datatype);  
38     console.log("Selected Tests:", selectedTests);  
39 }  
40  
41 // Clear button handler  
42 clearBtn.addEventListener("click", function () {  
43     urlInput.value = "";  
44     endpointInput.value = "";  
45     datatypeSelect.selectedIndex = 0;  
46     checkboxes.forEach(cb => cb.checked = false);  
47     showMessage("");  
48 });  
49  
50 // Save button handler  
51 saveBtn.addEventListener("click", async function () {  
52     const url = urlInput.value.trim();  
53     const endpoint = endpointInput.value.trim();  
54     const datatype = datatypeSelect.value;  
55     const selectedTests = Array.from(checkboxes)  
56     .filter(cb => cb.checked)  
57     .map(cb => cb.name);  
58  
59     if (!url || !endpoint) {  
60         showMessage("Please fill out the URL and Endpoint fields before saving.", true);  
61         return;  
62     }  
63     if (!isValidUrl(url)) {  
64         showMessage("Please enter a valid URL (e.g., https://example.com).", true);  
65         return;  
66     }  
67  
68     // Save job specification to the database  
69     const jobSpec = {  
70         url: url,  
71         endpoint: endpoint,  
72         datatype: datatype,  
73         tests: selectedTests  
74     };  
75     try {  
76         await fetch(`${API_URL}/api/jobs`, {  
77             method: 'POST',  
78             headers: {  
79                 'Content-Type': 'application/json',  
80             },  
81             body: JSON.stringify(jobSpec),  
82         });  
83         showMessage("Job specification saved successfully.", true);  
84     } catch (error) {  
85         console.error("Error saving job specification:", error);  
86         showMessage("Error saving job specification. Please try again.", true);  
87     }  
88 }  
89  
90 // Add new test button handler  
91 addTestBtn.addEventListener("click", function () {  
92     const testName = testNameInput.value.trim();  
93     if (!testName) {  
94         showMessage("Please enter a test name.", true);  
95         return;  
96     }  
97     const testId = `test-${Date.now()}`;  
98     const test = {  
99         name: testName,  
100        id: testId  
101    };  
102    try {  
103        fetch(`${API_URL}/api/tests`, {  
104            method: 'POST',  
105            headers: {  
106                'Content-Type': 'application/json',  
107            },  
108            body: JSON.stringify(test),  
109        });  
110        showMessage("Test added successfully.", true);  
111    } catch (error) {  
112        console.error("Error adding test:", error);  
113        showMessage("Error adding test. Please try again.", true);  
114    }  
115 }  
116  
117 // Run tests button handler  
118 runTestsBtn.addEventListener("click", function () {  
119     // Run tests logic (not implemented)  
120     showMessage("Running tests (not implemented).", true);  
121 }  
122  
123 // Logout button handler  
124 logoutBtn.addEventListener("click", function () {  
125     // Logout logic (not implemented)  
126     showMessage("Logging out (not implemented).", true);  
127 }  
128  
129 // Initialize the page  
130 document.addEventListener('DOMContentLoaded', function () {  
131     // Initialize the form fields  
132     urlInput.value = "https://api.example.com/data";  
133     endpointInput.value = "/v1/products";  
134     datatypeSelect.value = "JSON";  
135     // Initialize the test checkboxes  
136     checkboxes.forEach(cb => cb.checked = false);  
137     // Initialize the message box  
138     showMessage("", false);  
139 }  
140
```

Back-End

- Structure: models, routes, controllers, and src components are split into respective folders – ease for collaboration
- Frontend integration: clear /api/jobs endpoints for each of the job operations with straightforward responses and formats
- Challenge: validation and user feedback for each of the job spec fields on the front end, especially for back end error handling
 - Clearly validation logic and user messages

```
1  const Job = require('../models/jobModel');
2
3  // GET /api/jobs
4  exports.getAllJobs = (req, res) => {
5    res.json(Job.getAll());
6  };
7
8  // GET /api/jobs/:id
9  exports.getJobById = (req, res) => {
10   const job = Job.getById(req.params.id);
11   if (!job) return res.status(404).json({ error: 'Job not found' });
12   res.json(job);
13 };
14
15 // POST /api/jobs
16 exports.createJob = (req, res) => {
17   const { url, endpoint, datatype, selectedTests } = req.body;
18   if (!url || !endpoint) return res.status(400).json({ error: 'URL and endpoint required' });
19   try { new URL(url); } catch { return res.status(400).json({ error: 'Invalid URL' }); }
20   if (!endpoint.startsWith('/') || /\s/.test(endpoint)) {
21     return res.status(400).json({ error: 'Endpoint must start with "/" and contain no spaces'
22   });
23   const job = Job.create({ url, endpoint, datatype, selectedTests, createdAt: new Date().toISC
24   res.status(201).json(job);
25 };
26
27 // PUT /api/jobs/:id
28 exports.updateJob = (req, res) => {
29   const job = Job.update(req.params.id, req.body);
30   if (!job) return res.status(404).json({ error: 'Job not found' });
31   res.json(job);
32 };
33
34 // DELETE /api/jobs/:id
35 exports.deleteJob = (req, res) => {
36   const ok = Job.delete(req.params.id);
37   if (!ok) return res.status(404).json({ error: 'Job not found' });
38   res.status(204).end();
39 };
```

Challenges & Insights

- Again, biggest challenge this milestone was ensuring that the data exchanged amongst the frontend and backend was the same format, and having validation to ensure that. Insight from this was the importance of clear API contracts via validation, and how to set that up in a full environment. Dev tools and console.log statements helped make this process a lot faster and easier

Future Improvements & Next Steps

- Allowing for the actual implementation of running the vulnerability scanner
- Testing for bugs

Individual Team Member Slides

-Mason Lemberger-

Assigned Work Summary

- **Work done this milestone:** Implemented backend js for history and analytics page:
 - Created the controller, model, and route for the analytics page
 - Made two get requests and one delete
 - Updated front-end logic to work with the newly created api
- + Issue (closed) link: <https://github.com/MrLarryMan/Tensile/issues/36>
- + PR (merged) link: <https://github.com/MrLarryMan/Tensile/pull/43>

- Status: this milestone dealt with the frontend to backend js part, which I have completed. → Completed & closed all issues in this milestone
- Links to all issues you are assigned in the milestone that were not closed: none

Feature Demonstration

Frontend:

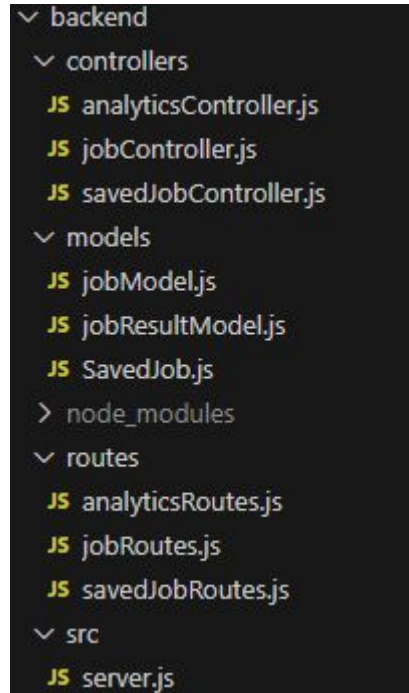
- Have multiple fetches to backend api
- Logic that utilizes sessionStorage to reduce api calls

Backend:

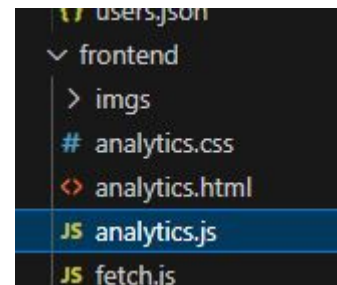
- Created gets and delete http requests for to manage analytics page
- Also added create in the model section so other part of the webpage can fill in the database

Code Structure & Organization

- Code structure: this follows the class structure of controller, models, and routes
- The front-end also have clearly labeled files to help understand the path the data takes
- All of the backend files are under the backend folder and the frontend files are under the frontend folder to help with clarity
- Critical components: all functionality relating to the analytics page is named after the page; the jobResultModel is also separated from the other model to be clearly defined



```
▼ backend
  ▼ controllers
    JS analyticsController.js
    JS jobController.js
    JS savedJobController.js
  ▼ models
    JS jobModel.js
    JS jobResultModel.js
    JS SavedJob.js
  > node_modules
  ▼ routes
    JS analyticsRoutes.js
    JS jobRoutes.js
    JS savedJobRoutes.js
  ▼ src
    JS server.js
```



```
▼ frontend
  > imgs
  # analytics.css
  <> analytics.html
  JS analytics.js
  JS fetch.js
```


Front-end

- Have multiple functions that ask the backend api for information
- This data is then present to the use using js

```
export async function getJobData(jobID) {
  try {
    const response = await fetch(`${ANALYTICS_BASE_URL}/${jobID}`);
    if (!response.ok) {
      throw new Error('Response status ${response.status}');
    }

    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Fetch failed at ${ANALYTICS_BASE_URL}/${jobID}', error);
    return null; // Return null on error
  }
}

export async function deleteJobData(jobID) {
  try {
    const response = await fetch(`${ANALYTICS_BASE_URL}/${jobID}`, {
      method: 'DELETE',
    });
    if (!response.ok) {
      throw new Error('Response status ${response.status}');
    }

    return true;
  } catch (error) {
    console.error('Fetch failed at ${ANALYTICS_BASE_URL}/${jobID}', error);
    return false; // Return false on error
  }
}
```

```
addEventListener("DOMContentLoaded", async () => {

  updatePage();

  const failedTestsSelections = document.getElementById("failed-tests");
  failedTestsSelections.addEventListener("change", async () => {
    updateFailedTestInfo(parseInt(failedTestsSelections.value));
  });

  const jobSelection = document.getElementById("Past Jobs");
  jobSelection.addEventListener("change", async () => {
    refreshJobInfo(parseInt(jobSelection.value));
  });

  const deleteJobButton = document.getElementById("delete-job-btn");
  deleteJobButton.addEventListener("click", async () => {
    deleteJob(parseInt(jobSelection.value));
  });

});
```

Back-end

- Created the controller, routes, and model for the backend
- This data is then present to the use using js

```
exports.create = (jobResultData) => {  
  const jobResults = readJobResults();  
  const newJobResult = { ...jobResultData, id: Date.now().toString() };  
  jobResults.push(newJobResult);  
  writeJobResults(jobResults);  
  return newJobResult;  
};  
  
exports.update = (id, jobResultData) => {  
  const jobResults = readJobResults();  
  const idx = jobResults.findIndex(jobResult => jobResult.id === id.toString());  
  if (idx === -1) return null;  
  jobResults[idx] = { ...jobResults[idx], ...jobResultData };  
  writeJobResults(jobResults);  
  return jobResults[idx];  
};  
  
exports.deleteById = (id) => {  
  const jobResults = readJobResults();  
  const idx = jobResults.findIndex(jobResult => jobResult.id === id.toString());  
  if (idx === -1) return false;  
  jobResults.splice(idx, 1);  
  writeJobResults(jobResults);  
  return true;  
};
```

You, 4 hours ago via PR #42 • new model created

```
const express = require('express');  
const router = express.Router();  
const analyticsController = require('../controllers/analyticsController');  
  
router.get('/', analyticsController.getJobResults);  
router.get('/:id', analyticsController.getJobResultById);  
router.delete('/:id', analyticsController.deleteJobResultById);  
  
module.exports = router;
```

You, 4 hours ago via PR #42 • Create the route

Challenges & Insights

- Obstacles faced and lessons acquired: One challenge I faced was figuring out how to implement code that heavily relies on other team members. This expanded the way I thought about coding and how to make my code more approachable to other members of the project.
- Key takeaways from working within a collaborative team environment: Once your code starts communicating across the different sections, it becomes very important to be very detailed oriented when discussing future plans and how work needs to be delegated. If this doesn't happen it can quickly become confusing

Future Improvements & Next Steps

- Future improvements or features: Finishing up any loose ends as well as making sure that all connections across the website function as needed.
- Technical debt or aspects that could be optimized further: none for now.
- Future Issue link: <https://github.com/MrLarryMan/Tensile/issues/20>

Individual Team Member Slides

-Larry Liu-

Assigned Work Summary

- **Work done this milestone:** Implemented backend Javascript code for vulnerability scanning (specifically XSS and LFI tests) and API endpoints to trigger vulnerability scans, integrating with job result and specification models.
 - Created a controller and route to interface with scanning functionality, as well as 3 code files in the backend/src directory.
 - Made One Get request, 2 Post request, and 1 Delete API endpoint. Backend vulnerability scanning code also makes use of all possible requests.
 - Updated front end logic of multiple views, including job specification and saved jobs, to make requests to the scanning API.
- + Issue (closed) link: <https://github.com/MrLarryMan/Tensile/issues/37?issue=MrLarryMan%7CTensile%7C39>
- + PR (merged) link: <https://github.com/MrLarryMan/Tensile/pull/43>
- Status: The milestone covers backend features, which have been completed → Completed & closed all issues in this milestone
- Non-closed Issues: N/A

Feature Demonstration

Frontend Integration:

- Modified the run functions in Job Specification.js and Saved Jobs.js to make requests to the scanner API.

Backend Integration:

- Created backend code that interfaces with the Job Results and Saved Job models to create and run jobs, as well as saving, modifying, and deleting job results.
- Made 1 GET, 2 POST, and 1 Delete API endpoint.
- Utilized GET, POST, PUT, DELETE requests in XSS and LFI scanning code in advanced ways.

Code Explanation:

- Code Structure: This follows the class structure of controllers, models, and routes. My constitution did not make its own model as its database interactions would have been redundant, but instead integrated existing models.
- All of the backend files are under the backend folder and the frontend files are under the frontend folder to help with clarity. I have contributions specifically in the routes, controller, and backend/src folders. Also light modifications in the frontend folder.
- Critical components: all code containing functionality relating to the scanning functionality is denoted with scanner in the name.
- The scanning is primarily handled through the scannerMain function, which imports other javascript files that contain the code for scans of specific vulnerability types. Tools used were node puppeteer for XSS and fetch for LFI.

```
exports.xssScan = async (url, reqType, dataType) => {  
  // A polygot XSS payload, capable of getting past many XSS filters  
  let payload = encodeURIComponent("jaVasCript:/*-/*/*'\`/*/*/*/*/*/*/*/*oNcliCk=alert() )//%0D%0A%0d%0a/#!/stYl");  
  if(dataType === "base64") {  
    payload = btoa(payload);  
  } else if (dataType === "JSON") {  
    // todo: add an option for them to provide a name/key to the data.  
    payload = json.loads("{\"data: " + payload + "}");  
  }  
  
  return new Promise(async (res, rej) => {  
    const browser = await puppeteer.launch({  
      executablePath: '/usr/bin/google-chrome-stable',  
      args: ['--no-sandbox']  
    });  
  });  
}
```

```
<> job_specification.html  
JS job_specification.js  
# login-styles.css  
<> login.html  
JS login.js  
<> register.html  
JS register.js  
<> saved_job.html  
JS saved_jobs.js
```

```
exports.runJob = (jobId) => {  
  const job = SavedJob.getById(jobId);  
  const tests = job.test_options;  
  for (const key in tests) {  
    if (tests.hasOwnProperty(key)) { // Ensure it's not from the prototype chain  
      if (key === "XSS" && tests[key] === true) {  
        let template = createJobResultTemplate(job);  
        const result = jobResultModel.create(template);  
        scannerXSS.xssScan(job.url + job.endpoint, job.reqType, job.dataType).then((vulns) => {  
          template.vulns = vulns;  
          jobResultModel.update(result.id, template);  
        }).catch((error) => {  
          template.status = "Failed";  
          jobResultModel.update(result.id, template);  
        });  
        return result.id;  
      } else if (key === "LFI" && tests[key] === true) {  
        let template = createJobResultTemplate(job);  
        const result = jobResultModel.create(template);  
        scannerLFI.LFIScan(job.url + job.endpoint, job.reqType, job.dataType).then((vulns) => {  
          template.status = "Finished";  
        });  
      }  
    }  
  }  
}
```


Front-end

- Hooked up all frontend code that runs jobs so that they call the scanner API endpoints
- After jobs are run, the user is notified of the progress and can navigate to the results/analysis page to see the results of the job.

```
function handleRunJob() {  
  if (!selectedJobId) {  
    alert("Please select a job first");  
    return;  
  }  
  //alert(`Running job: ${selectedJobId}`);  
  
  // URL for testing. Port number may change in dev or production.  
  fetch(`127.0.0.1:3000/runJob?jobId=${selectedJobId}`)  
}
```

```
const options = {  
  method: 'POST',  
  body: {  
    url: url,  
    endpoint: endpoint,  
    request_type: requestType,  
    test_options: selectedTests,  
    datatype: datatype  
  }  
};  
  
// URL for testing. Port number may change in dev or production.  
fetch("127.0.0.1:3000/createAndRunJob", options)
```

Back-end

- Created the controller, routes, and backend code for scanner functionality (a main file and individual files for different tests.)
- The data is stored upon jobs being run. The user is provided with the job result ids so they can review with additional requests.

```
const express = require('express');
const router = express.Router();
const scannerController = require('../controllers/scannerController');

router.get('/:resultId', scannerController.jobStatus);
router.post('/', scannerController.createAndRunJob);
router.post('/:jobId', scannerController.runJob);
router.delete('/:resultId', scannerController.terminateJob);

module.exports = router;
```

```
exports.runJob = (jobId) => {
  const job = SwedenJob.getById(jobId);
  const tests = job.test_options;
  for (const key in tests) {
    if (tests.hasOwnProperty(key)) { // Ensure it's not from the prototype chain
      if (key === 'XSS' && tests[key] == true) {
        let template = createJobResultTemplate(job);
        const result = jobResultModel.create(template);
        scannerXSS.xssScan(job.url + job.endpoint, job.requestType, job.dataType).then((vulns) => {
          template.status = 'Finished';
          template.vulns = vulns;
          jobResultModel.update(result.id, template);
        }).catch((error) => {
          template.status = 'Failed';
          jobResultModel.update(result.id, template);
        });
      }
      return result.id;
    } else if (key === 'i1' && tests[key] == true) {
      let template = createJobResultTemplate(job);
      const result = jobResultModel.create(template);
      scannerI1.i1Scan(job.url + job.endpoint, job.requestType, job.dataType).then((vulns) => {
        template.status = 'Finished';
      });
    }
  }
}
```

```
// Does not have its own model, as it makes use of the jobResult and SavedJob models in its service code.
const scannerMain = require('../src/scannerMain')

// GET, returns the status of the job (Running, Failed, or Finished)
exports.jobStatus = async (req, res) => {
  try {
    const jobStatus = await scannerMain.jobStatus(req.params.resultId);
    res.status(200).json(jobStatus);
  } catch (error) {
    res.status(500).json({ message: 'Error fetching job status', error });
  }
};

// POST, runs a job by its id. Will return the job result id (not necessarily finished running).
exports.runJob = async (req, res) => {
  try {
    const jobResult = await scannerMain.runJob(req.params.resultId);
    res.status(200).json(jobResult);
  } catch (error) {
    res.status(500).json({ message: 'Error running job', error });
  }
};

// POST, runs a creates and runs a job with specs taken from the body of the request.
// Will return the job result id (not necessarily finished running).
exports.createAndRunJob = async (req, res) => {
  try {
    const jobResult = await scannerMain.createAndRunJob(req.body);
    res.status(200).json(jobResult);
  } catch (error) {
    res.status(500).json({ message: 'Error running job', error });
  }
};

// DELETE, Terminates a running job with its result id and deletes its results (only if job did not finish)
```

Challenges & Insights

- Obstacles faced and lessons acquired: As the tests were implemented, I realized that each individual test varied significantly more based on the context of the application than anticipated. We did not add sufficient customizability for the full suite of initially planned features. I learned that one should always anticipate the amount of detail required for the implementation of a feature to increase as it is worked on, and that you should account for this by being flexible with the design of your tool.
- Key takeaways from working within a collaborative team environment: It's important for your work to be understandable by others and easy to hand off, because you cannot rely on always being able to stay on a specific assignment as more and different work arises.

Future Improvements & Next Steps

- Future improvements or features: Implement additional tests, test types, and customizability for tests. Assure that jobs are properly saved on frontend endpoints and minimize redundancy.
- Technical debt or aspects that could be optimized further: See above. We also have some minor conflicting coding conventions / naming conventions in our routes/models which may need polishing..
- Issue link: <https://github.com/MrLarryMan/Tensile/issues/37>