PITSTACK

COMPREHENSIVE DEVELOPMENT PROGRAM

12-Week Professional Learning Path

What Makes This Plan Different

✓ Every project builds your actual product
✓ Daily tasks with specific time estimates
✓ Professional development practices from Day 1
✓ Expert recommendations at every step

Version 3.0 — Professional Edition

Table of Contents

Introduction: Your Path to Building PITSTACK

This is not a typical programming tutorial. This is a structured professional development program designed specifically for you—someone who knows programming basics but wants to learn by building something real, not copying AI-generated code.

The Philosophy Behind This Plan

Most learning paths make a critical mistake: they teach concepts using throwaway projects. You build a weather app to learn APIs, a todo app to learn databases, then throw them all away when you finally start your 'real' project.

This plan is different. Every single line of code you write becomes part of PITSTACK. The CVE fetcher you build in Week 1 becomes your production CVE collector. The database schema you design in Week 4 is the actual database your users will query.

MY RECOMMENDATION

Treat this like a job, not a hobby. Set specific hours for learning (I recommend 2-3 hours daily). Show up even when you don't feel like it. Consistency beats intensity every time.

How to Use This Document

- Follow the weeks in order—each builds on the previous
- Complete daily tasks before moving to the next day
- Use the checkboxes to track your progress
- Read the 'My Recommendation' boxes—they contain hard-won wisdom
- Don't skip the 'Why This Matters' sections
- Take notes in a separate document or Notion

Time Commitment

Schedule	Daily Hours	Weekly Total
Minimum	2 hours	14 hours
Recommended	**3 hours**	**21 hours**
Intensive	4-5 hours	28-35 hours

Pre-Work: Setup Before Week 1

Complete these tasks before starting Week 1. This ensures you can focus on learning, not troubleshooting installations.

Development Environment Setup

1. Install Node.js

- Go to nodejs.org and download the LTS (Long Term Support) version
- Run the installer with default settings
- Verify installation: open terminal, type 'node --version' (should show v18+ or v20+)
- Verify npm: type 'npm --version'

2. Install VS Code

- Download from code.visualstudio.com
- Install these essential extensions:
    - ESLint - catches errors as you type
    - Prettier - formats your code automatically
    - JavaScript (ES6) code snippets
    - GitLens - shows git history in editor
    - Thunder Client - test APIs without leaving VS Code

3. Install Git

- Download from git-scm.com
- During installation, choose VS Code as default editor
- Verify: 'git --version' in terminal
- Configure your identity:

```
Git Configuration
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

4. Create Accounts

- GitHub account (github.com) - for code hosting
- NVD API Key (nvd.nist.gov/developers/request-an-api-key) - higher rate limits

5. Create Project Structure

```
Initial Project Setup
mkdir pitstack
cd pitstack
npm init -y
git init
mkdir collectors database scoring api frontend auth realtime notifications
```

```
touch .gitignore README.md
```

```
.gitignore Contents
node_modules/
.env
.DS_Store
*.log
dist/
coverage/
```

Don't Skip This

Spending 1-2 hours on proper setup saves 10+ hours of frustration later. If something doesn't install correctly, Google the exact error message—someone has solved it before.

WEEK 1

CVE Vulnerability Collector

Learn JavaScript fundamentals by fetching real security data

Week Overview

This week you build the foundation of PITSTACK's vulnerability intelligence. By Day 7, you'll have a working CLI tool that fetches CVE data from the National Vulnerability Database—the same data source used by enterprise security tools.

Why This Matters

CVEs (Common Vulnerabilities and Exposures) are the backbone of vulnerability management. Every security team tracks CVEs. Your tool will aggregate this data automatically, saving hours of manual checking.

Skills You'll Gain

- Async JavaScript (the most important JS concept)
- Working with REST APIs
- Parsing and transforming JSON data
- Command-line application design
- File I/O for data persistence

DAY 1: Environment Verification & JavaScript Refresher

Morning Session (1.5 hours): Verify Setup

Task	Time	Status
Verify Node.js installation (node --version)	5 min	☐
Verify npm installation (npm --version)	5 min	☐
Open pitstack folder in VS Code	5 min	☐
Create collectors/test.js with console.log	10 min	☐
Run with 'node collectors/test.js'	5 min	☐
Install nodemon globally: npm install -g nodemon	10 min	☐
Test nodemon: nodemon collectors/test.js	10 min	☐

☐ What is Nodemon?

Nodemon automatically restarts your script when you save changes. Instead of typing 'node file.js' every time, nodemon watches for changes and re-runs automatically. Essential for development.

Afternoon Session (1.5 hours): JavaScript Fundamentals

Create collectors/js-practice.js and practice these concepts using cybersecurity-themed examples:

```
Variables and Data Types
// Variables - use const by default, let when you need to reassign
const cveId = "CVE-2024-1234";
const severity = "CRITICAL";
let attacksDetected = 0;

// Objects - you'll use these constantly
const vulnerability = {
  id: "CVE-2024-1234",
  description: "Remote code execution in Apache",
  cvssScore: 9.8,
  severity: "CRITICAL",
  publishedDate: "2024-01-15"
};

// Accessing object properties
console.log(vulnerability.id);           // CVE-2024-1234
console.log(vulnerability["severity"]);  // CRITICAL

// Arrays - lists of items
const recentCVEs = [
  { id: "CVE-2024-001", score: 9.8 },
  { id: "CVE-2024-002", score: 7.5 },
  { id: "CVE-2024-003", score: 4.2 }
];
```

```
Array Methods (You'll Use These Daily)
// filter - keep items that match a condition
const criticalCVEs = recentCVEs.filter(cve => cve.score >= 9.0);

// map - transform each item
const cveIds = recentCVEs.map(cve => cve.id);

// find - get first match
const specificCVE = recentCVEs.find(cve => cve.id === "CVE-2024-002");

// forEach - do something with each item
recentCVEs.forEach(cve => {
  console.log(`${cve.id}: ${cve.score}`);
});
```

Evening Task: Self-Assessment

Without looking at examples, try to:

- ☐ Create an object representing a security news article
- ☐ Create an array of 5 such objects
- ☐ Filter to only articles from 'today'
- ☐ Map to get just the titles

MY RECOMMENDATION

If you can't complete the self-assessment without help, spend another hour on javascript.info chapters 2, 4, and 5 before moving to Day 2. These fundamentals are critical.

DAY 2: Async JavaScript & Promises

Why Async Matters

When you fetch data from an API, your code doesn't wait—it continues executing while the network request happens in the background. This is asynchronous programming, and it's how all modern JavaScript applications work.

Morning Session (1.5 hours): Understanding Promises

The Problem: Network Requests Take Time

```
// This WON'T work as you might expect
const data = fetch("https://api.example.com/data");
console.log(data); // Promise { <pending> } - NOT the actual data!

// Why? fetch() returns immediately with a Promise
// The actual data arrives later
```

Solution 1: Promise .then() Chains

```
fetch("https://api.example.com/data")
  .then(response => response.json())  // Convert response to JSON
  .then(data => {
    console.log(data);  // NOW we have the actual data
  })
  .catch(error => {
    console.error("Something went wrong:", error);
  });
```

Solution 2: async/await (Preferred)

```
// async/await makes async code look synchronous
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);  // Clean and readable!
    return data;
  } catch (error) {
    console.error("Something went wrong:", error);
  }
}

// Call the async function
fetchData();
```

async/await is Just Syntax Sugar

async/await is built on Promises—it's just a cleaner way to write them. Use async/await for most code, but understand Promises because you'll see them in documentation and libraries.

Afternoon Session (1.5 hours): Your First API Call

Let's fetch real data. Create collectors/cve-collector.js:

First CVE Fetch

```
// collectors/cve-collector.js

async function fetchCVEs() {
  const baseUrl = "https://services.nvd.nist.gov/rest/json/cves/2.0";

  // Get CVEs from the last 7 days
  const endDate = new Date().toISOString();
  const startDate = new Date(Date.now() - 7 * 24 * 60 * 60 * 1000).toISOString();

  const url = `${baseUrl}?pubStartDate=${startDate}&pubEndDate=${endDate}&resultsPerPage=10`;

  console.log("Fetching CVEs from NVD...");
  console.log("URL:", url);

  try {
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const data = await response.json();

    console.log(`Found ${data.totalResults} total CVEs`);
    console.log(`Showing first ${data.vulnerabilities.length}\n`);

    // Process each CVE
    data.vulnerabilities.forEach(item => {
      const cve = item.cve;
      const id = cve.id;
      const description = cve.descriptions.find(d => d.lang === "en")?.value || "No
description";

      console.log(`${id}`);
      console.log(`  ${description.substring(0, 100)}...`);
      console.log("");
    });

    return data;
  } catch (error) {
    console.error("Failed to fetch CVEs:", error.message);
  }
}

// Run it
fetchCVEs();
```

Evening Task: Experiment

- ☐ Run the script: node collectors/cve-collector.js
- ☐ Change resultsPerPage to 20
- ☐ Change the date range to last 30 days

- ☐ Add console.log statements to understand the data structure

Rate Limits

NVD API allows 5 requests per 30 seconds without an API key, or 50 requests per 30 seconds with a key. For development, this is fine. Add your API key in Week 2 when we implement scheduled fetching.

DAY 3: Parsing CVE Data & Extracting CVSS Scores

Understanding CVE Data Structure

CVE data from NVD is deeply nested. Today you'll learn to navigate this structure and extract the fields that matter for PITSTACK.

Morning Session: Deep Dive into CVE Structure

CVE Data Structure (Simplified)

```
{
  "vulnerabilities": [
    {
      "cve": {
        "id": "CVE-2024-1234",
        "descriptions": [
          { "lang": "en", "value": "Description text here" }
        ],
        "published": "2024-01-15T10:15:00.000",
        "lastModified": "2024-01-16T14:30:00.000",
        "metrics": {
          "cvssMetricV31": [
            {
              "cvssData": {
                "baseScore": 9.8,
                "baseSeverity": "CRITICAL",
                "vectorString": "CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H"
              }
            }
          ]
        },
        "weaknesses": [...],
        "references": [...]
      }
    }
  ]
}
```

Notice how metrics can have cvssMetricV31, cvssMetricV30, or cvssMetricV2. You need to handle all cases.

Robust CVSS Extraction

```
function extractCVSSScore(cve) {
  const metrics = cve.metrics;

  // Try CVSS 3.1 first (preferred)
  if (metrics?.cvssMetricV31?.[0]) {
    const cvss = metrics.cvssMetricV31[0].cvssData;
    return {
      version: "3.1",
      score: cvss.baseScore,
      severity: cvss.baseSeverity,
      vector: cvss.vectorString
    };
  }
  // Fall back to CVSS 3.0
  if (metrics?.cvssMetricV30?.[0]) {
    const cvss = metrics.cvssMetricV30[0].cvssData;
    return {
      version: "3.0",
      score: cvss.baseScore,
      severity: cvss.baseSeverity,
      vector: cvss.vectorString
    };
  }

  // Fall back to CVSS 2.0
  if (metrics?.cvssMetricV2?.[0]) {
    const cvss = metrics.cvssMetricV2[0].cvssData;
    return {
      version: "2.0",
      score: cvss.baseScore,
      severity: cvss.baseSeverity || getSeverityFromScore(cvss.baseScore),
      vector: cvss.vectorString
    };
  }

  // No CVSS data available
  return {
    version: null,
    score: null,
    severity: "UNKNOWN",
    vector: null
  };
}

function getSeverityFromScore(score) {
  if (score >= 9.0) return "CRITICAL";
  if (score >= 7.0) return "HIGH";
  if (score >= 4.0) return "MEDIUM";
  return "LOW";
}
```

Afternoon Session: Create Normalized Data Structure

Create a function that transforms raw NVD data into PITSTACK's standard format:

Data Normalization Function

```
function normalizeCVE(nvdItem) {
  const cve = nvdItem.cve;
  const cvss = extractCVSSScore(cve);

  return {
    // Standard PITSTACK fields
    id: cve.id,
    type: "cve",
    title: cve.id,  // CVEs use ID as title
    description: cve.descriptions.find(d => d.lang === "en")?.value || "",
    url: `https://nvd.nist.gov/vuln/detail/${cve.id}`,
    source: "nvd",
    publishedAt: new Date(cve.published),

    // CVE-specific metadata
    metadata: {
      cvss: cvss,
      weaknesses: cve.weaknesses?.map(w => w.description[0]?.value) || [],
      references: cve.references?.map(r => r.url) || [],
      lastModified: new Date(cve.lastModified)
    }
  };
}

// Usage
const normalizedCVEs = data.vulnerabilities.map(normalizeCVE);
```

Why Normalize Data?

RSS feeds, CTF events, and bug bounty programs all have different data structures. By normalizing everything to a standard format, your dashboard code doesn't need to handle special cases—it just displays items.

Evening Task: Update Your Collector

- ☐ Add extractCVSSScore() function to cve-collector.js
- ☐ Add normalizeCVE() function
- ☐ Update fetchCVEs() to return normalized data
- ☐ Log normalized data and verify all fields are populated

DAY 4: Command Line Arguments & User Input

Making Your Tool Configurable

A good CLI tool accepts arguments so users can customize behavior without editing code.

Morning Session: Processing Arguments

```
Basic Argument Parsing
// process.argv contains command line arguments
// [0] = node path, [1] = script path, [2+] = your arguments

// node script.js --limit 20 --severity critical
console.log(process.argv);
// ['node', '/path/script.js', '--limit', '20', '--severity', 'critical']

// Simple parser
function parseArgs() {
  const args = process.argv.slice(2);
  const options = {
    limit: 10,         // default
    severity: null,    // default: all
    keyword: null,     // default: none
    help: false
  };

  for (let i = 0; i < args.length; i++) {
    switch (args[i]) {
      case '--limit':
      case '-l':
        options.limit = parseInt(args[++i], 10);
        break;
      case '--severity':
      case '-s':
        options.severity = args[++i].toUpperCase();
        break;
      case '--keyword':
      case '-k':
        options.keyword = args[++i].toLowerCase();
        break;
      case '--help':
      case '-h':
        options.help = true;
        break;
    }
  }

  return options;
}
```

```
Help Message
function showHelp() {
  console.log(`
PITSTACK CVE Collector
======================
Usage: node cve-collector.js [options]

Options:
  -l, --limit <n>      Number of CVEs to fetch (default: 10)
  -s, --severity <s>   Filter by severity: LOW, MEDIUM, HIGH, CRITICAL
  -k, --keyword <k>    Filter by keyword in description
  -h, --help           Show this help message

Examples:
  node cve-collector.js --limit 20
  node cve-collector.js --severity critical
  node cve-collector.js -l 50 -s high -k apache
  `);
}
```

Afternoon Session: Implement Filtering

Filter Functions

```
function filterBySeverity(cves, severity) {
  if (!severity) return cves;
  return cves.filter(cve => cve.metadata.cvss.severity === severity);
}

function filterByKeyword(cves, keyword) {
  if (!keyword) return cves;
  const lowerKeyword = keyword.toLowerCase();
  return cves.filter(cve =>
    cve.description.toLowerCase().includes(lowerKeyword) ||
    cve.id.toLowerCase().includes(lowerKeyword)
  );
}

// Main function with options
async function main() {
  const options = parseArgs();

  if (options.help) {
    showHelp();
    return;
  }

  console.log("Fetching CVEs with options:", options);

  let cves = await fetchCVEs(options.limit * 2);  // Fetch extra for filtering

  cves = filterBySeverity(cves, options.severity);
  cves = filterByKeyword(cves, options.keyword);
  cves = cves.slice(0, options.limit);  // Apply final limit

  displayCVEs(cves);

  return cves;
}
```

MY RECOMMENDATION

Consider using a library like 'commander' or 'yargs' for complex CLI apps. But for PITSTACK, manual parsing teaches you how it works and keeps dependencies minimal.

Evening Task: Test Your CLI

- ☐ Test: node cve-collector.js --help
- ☐ Test: node cve-collector.js --limit 5
- ☐ Test: node cve-collector.js --severity critical
- ☐ Test: node cve-collector.js -l 10 -s high -k microsoft

DAY 5: File I/O & Data Persistence

Why Persistence Matters

Without saving data, your tool forgets everything between runs. Today you'll add the ability to save CVEs to a file and detect which ones are new.

Morning Session: File System Basics

```
Reading and Writing Files
const fs = require('fs');
const path = require('path');

// Ensure data directory exists
const dataDir = path.join(__dirname, '..', 'data');
if (!fs.existsSync(dataDir)) {
  fs.mkdirSync(dataDir, { recursive: true });
}

const cveFilePath = path.join(dataDir, 'cves.json');

// Save CVEs to file
function saveCVEs(cves) {
  const data = {
    lastUpdated: new Date().toISOString(),
    count: cves.length,
    items: cves
  };

  fs.writeFileSync(cveFilePath, JSON.stringify(data, null, 2));
  console.log(`Saved ${cves.length} CVEs to ${cveFilePath}`);
}

// Load CVEs from file
function loadCVEs() {
  if (!fs.existsSync(cveFilePath)) {
    return { lastUpdated: null, count: 0, items: [] };
  }

  const content = fs.readFileSync(cveFilePath, 'utf-8');
  return JSON.parse(content);
}
```

Afternoon Session: Detecting New CVEs

```
Change Detection
function detectNewCVEs(freshCVEs, existingData) {
  const existingIds = new Set(existingData.items.map(cve => cve.id));

  const newCVEs = freshCVEs.filter(cve => !existingIds.has(cve.id));
  const updatedCVEs = freshCVEs.filter(cve => existingIds.has(cve.id));

  return {
    new: newCVEs,
    existing: updatedCVEs,
    summary: {
      total: freshCVEs.length,
      newCount: newCVEs.length,
      existingCount: updatedCVEs.length
    }
  };
}

// Enhanced main function
async function main() {
  const options = parseArgs();

  // Load existing data
  const existingData = loadCVEs();
  console.log(`Loaded ${existingData.count} existing CVEs`);

  // Fetch fresh data
  const freshCVEs = await fetchCVEs(options.limit);

  // Detect changes
  const changes = detectNewCVEs(freshCVEs, existingData);

  console.log("\n=== CHANGE SUMMARY ===");
  console.log(`New CVEs: ${changes.summary.newCount}`);
  console.log(`Existing: ${changes.summary.existingCount}`);

  if (changes.new.length > 0) {
    console.log("\n□ NEW CVEs DETECTED:");
    changes.new.forEach(cve => {
      console.log(`  ${cve.id} - ${cve.metadata.cvss.severity}`);
    });
  }

  // Merge and save
  const allCVEs = mergeCVEs(freshCVEs, existingData.items);
  saveCVEs(allCVEs);
}

function mergeCVEs(fresh, existing) {
  const merged = new Map();

  // Add existing (will be overwritten by fresh if same ID)
  existing.forEach(cve => merged.set(cve.id, cve));

  // Add/update with fresh
  fresh.forEach(cve => merged.set(cve.id, cve));

  // Sort by date (newest first)
  return Array.from(merged.values())
    .sort((a, b) => new Date(b.publishedAt) - new Date(a.publishedAt));
}
```

□ This is Real Value

Enterprise security tools charge thousands of dollars for exactly this: fetching vulnerabilities, tracking what's new, and alerting on changes. You're building the core of that system.

Evening Task: Test Persistence

- ☐ Run collector: data/cves.json should be created
- ☐ Run again: should show 'Loaded X existing CVEs'
- ☐ Wait a day, run again: should detect new CVEs

DAY 6-7: Output Formatting & Code Organization

Day 6: Professional Output Formatting

Colored Console Output

```
// Terminal colors (ANSI escape codes)
const colors = {
  reset: "\x1b[0m",
  bright: "\x1b[1m",
  red: "\x1b[31m",
  green: "\x1b[32m",
  yellow: "\x1b[33m",
  blue: "\x1b[34m",
  cyan: "\x1b[36m"
};

function severityColor(severity) {
  switch (severity) {
    case 'CRITICAL': return colors.red;
    case 'HIGH': return colors.yellow;
    case 'MEDIUM': return colors.blue;
    case 'LOW': return colors.green;
    default: return colors.reset;
  }
}

function displayCVE(cve) {
  const color = severityColor(cve.metadata.cvss.severity);
  const score = cve.metadata.cvss.score?.toFixed(1) || 'N/A';

  console.log(`${colors.bright}${cve.id}${colors.reset}`);
  console.log(`  Score: ${color}${score} (${cve.metadata.cvss.severity})${colors.reset}`);
  console.log(`  ${cve.description.substring(0, 100)}...`);
  console.log(`  URL: ${colors.cyan}${cve.url}${colors.reset}`);
  console.log("");
}
```

Day 7: Code Organization

Refactor your code into clean, modular files:

File Structure After Week 1

```
collectors/
├── cve-collector.js    # Main entry point
├── lib/
│   ├── nvd-api.js      # API fetching logic
│   ├── normalizer.js   # Data normalization
│   ├── storage.js      # File I/O operations
│   ├── filters.js      # Filtering functions
│   └── display.js      # Output formatting
└── config.js           # Configuration constants
```

Example: lib/nvd-api.js

```
// lib/nvd-api.js
const config = require('../config');
```

```
async function fetchFromNVD(options = {}) {
  const {
    limit = 20,
    startDate = getDefaultStartDate(),
    endDate = new Date().toISOString()
  } = options;

  const url = buildNVDUrl({ startDate, endDate, limit });

  const response = await fetch(url, {
    headers: config.NVD_API_KEY
      ? { 'apiKey': config.NVD_API_KEY }
      : {}
  });

  if (!response.ok) {
    throw new Error(`NVD API error: ${response.status}`);
  }

  return response.json();
}

function buildNVDUrl({ startDate, endDate, limit }) {
  const params = new URLSearchParams({
    pubStartDate: startDate,
    pubEndDate: endDate,
    resultsPerPage: limit.toString()
  });

  return `${config.NVD_BASE_URL}?${params}`;
}

function getDefaultStartDate() {
  const date = new Date();
  date.setDate(date.getDate() - 7);
  return date.toISOString();
}

module.exports = { fetchFromNVD };
```

□ MODULE: cve-collector.js (Complete)

A full-featured CVE collection tool with CLI arguments, persistent storage, change detection, and professional output formatting.

Deliverable: collectors/cve-collector.js + collectors/lib/*.js + data/cves.json

Time: *~15 hours total*

Week 1 Completion Checklist

- ☐ CVE collector fetches data from NVD API
- ☐ Data is normalized to PITSTACK format
- ☐ CLI accepts --limit, --severity, --keyword, --help
- ☐ Data persists to data/cves.json

- ☐ New CVEs are detected and highlighted
- ☐ Code is organized into modules
- ☐ README.md documents usage
- ☐ Code is committed to Git with meaningful messages

WEEK 2

RSS Security News Collector

Parse multiple data formats and handle real-world API challenges

Week Overview

This week you expand PITSTACK's intelligence gathering to include security news from RSS feeds. You'll learn to parse XML (RSS format), handle multiple data sources, and deal with unreliable external services.

Skills You'll Gain

- XML/RSS parsing
- Promise.all for parallel requests
- Error handling for unreliable services
- Data deduplication
- Working with multiple data sources

DAY 1: Understanding RSS Feeds

What is RSS?

RSS (Really Simple Syndication) is an XML-based format for publishing frequently updated content. Most news sites, blogs, and podcasts offer RSS feeds. For cybersecurity, RSS is goldmine of real-time intelligence.

Security News RSS Feeds to Use

Source	RSS URL
The Hacker News	`https://feeds.feedburner.com/TheHackersNews`
Bleeping Computer	`https://www.bleepingcomputer.com/feed/`
Krebs on Security	`https://krebsonsecurity.com/feed/`
Dark Reading	`https://www.darkreading.com/rss.xml`
Threatpost	`https://threatpost.com/feed/`
Security Week	`https://www.securityweek.com/feed`

Installing RSS Parser

```
Setup
cd pitstack
npm install rss-parser
```

Basic RSS Fetching

```
// collectors/rss-collector.js
const Parser = require('rss-parser');
const parser = new Parser();

async function fetchRSSFeed(url, sourceName) {
  try {
    const feed = await parser.parseURL(url);

    console.log(`${sourceName}: ${feed.items.length} items`);

    return feed.items.map(item => ({
      id: item.guid || item.link,
      type: 'news',
      title: item.title,
      description: item.contentSnippet || item.content || '',
      url: item.link,
      source: sourceName,
      publishedAt: new Date(item.pubDate || item.isoDate),
      metadata: {
        author: item.creator || item.author,
        categories: item.categories || []
      }
    }));
  } catch (error) {
    console.error(`Error fetching ${sourceName}: ${error.message}`);
    return [];  // Return empty array, don't crash
  }
}

// Test it
fetchRSSFeed('https://feeds.feedburner.com/TheHackersNews', 'The Hacker News')
  .then(items => console.log(items.slice(0, 2)));
```

RSS Feeds Are Unreliable

Feeds go down, change URLs, return malformed XML, or block requests. Your code must handle all these cases gracefully. Never let one broken feed crash your entire collector.

DAY 2-3: Multi-Source Fetching with Promise.all

Fetching Multiple Sources in Parallel

Fetching 6 feeds one at a time takes ~6 seconds. Fetching them in parallel takes ~1 second. Promise.all makes this easy.

Parallel Fetching

```
const RSS_SOURCES = [
  { name: 'The Hacker News', url: 'https://feeds.feedburner.com/TheHackersNews' },
  { name: 'Bleeping Computer', url: 'https://www.bleepingcomputer.com/feed/' },
  { name: 'Krebs on Security', url: 'https://krebsonsecurity.com/feed/' },
  { name: 'Dark Reading', url: 'https://www.darkreading.com/rss.xml' },
  { name: 'Threatpost', url: 'https://threatpost.com/feed/' },
  { name: 'Security Week', url: 'https://www.securityweek.com/feed' }
];

async function fetchAllFeeds() {
  console.log(`Fetching from ${RSS_SOURCES.length} sources...`);
  const startTime = Date.now();

  // Fetch all sources in parallel
  const results = await Promise.all(
    RSS_SOURCES.map(source =>
      fetchRSSFeed(source.url, source.name)
    )
  );

  // Flatten results into single array
  const allItems = results.flat();

  const elapsed = ((Date.now() - startTime) / 1000).toFixed(2);
  console.log(`Fetched ${allItems.length} items in ${elapsed}s`);

  return allItems;
}
```

Promise.allSettled for Better Error Handling

Promise.all fails if ANY promise fails. Promise.allSettled waits for all, regardless of success/failure:

Robust Parallel Fetching

```
async function fetchAllFeedsRobust() {
  const results = await Promise.allSettled(
    RSS_SOURCES.map(source =>
      fetchRSSFeed(source.url, source.name)
    )
  );

  const successfulFeeds = [];
  const failedFeeds = [];

  results.forEach((result, index) => {
    if (result.status === 'fulfilled') {
      successfulFeeds.push(...result.value);
    } else {
      failedFeeds.push({
        source: RSS_SOURCES[index].name,
        error: result.reason.message
      });
    }
  });

  if (failedFeeds.length > 0) {
    console.warn("\nFailed feeds:");
    failedFeeds.forEach(f => console.warn(`  - ${f.source}: ${f.error}`));
  }

  console.log(`\nSuccessfully fetched: ${successfulFeeds.length} items`);
  return successfulFeeds;
}
```

□ MY RECOMMENDATION

Always use Promise.allSettled for external API calls. One flaky feed shouldn't break your entire data pipeline. Log failures but continue with available data.

DAY 4-5: Deduplication & Data Quality

Why Deduplication Matters

News aggregators often syndicate the same story. Without deduplication, your feed shows the same article 5 times from different sources.

```
Deduplication Strategies
function deduplicateItems(items) {
  const seen = new Map();
  const unique = [];

  for (const item of items) {
    // Strategy 1: Exact URL match
    if (seen.has(item.url)) {
      continue;
    }

    // Strategy 2: Similar title detection
    const normalizedTitle = normalizeTitle(item.title);
    if (seen.has(normalizedTitle)) {
      // Keep the one with more content
      const existing = seen.get(normalizedTitle);
      if (item.description.length > existing.description.length) {
        // Replace with better version
        const index = unique.findIndex(u => u.id === existing.id);
        unique[index] = item;
        seen.set(normalizedTitle, item);
      }
      continue;
    }

    seen.set(item.url, item);
    seen.set(normalizedTitle, item);
    unique.push(item);
  }

  return unique;
}

function normalizeTitle(title) {
  return title
    .toLowerCase()
    .replace(/[^a-z0-9]/g, '')  // Remove non-alphanumeric
    .substring(0, 50);           // First 50 chars
}
```

Day 5: Source Reliability Tracking

```
Track Source Health
const sourceStats = new Map();

function updateSourceStats(sourceName, success, itemCount, responseTime) {
  const stats = sourceStats.get(sourceName) || {
    name: sourceName,
    totalFetches: 0,
    successfulFetches: 0,
    totalItems: 0,
    avgResponseTime: 0,
    lastFetch: null,
    lastError: null
  };

  stats.totalFetches++;
  if (success) {
    stats.successfulFetches++;
    stats.totalItems += itemCount;
  }
  stats.avgResponseTime =
    (stats.avgResponseTime * (stats.totalFetches - 1) + responseTime) / stats.totalFetches;
  stats.lastFetch = new Date();

  sourceStats.set(sourceName, stats);
}

function getSourceReliability(sourceName) {
  const stats = sourceStats.get(sourceName);
  if (!stats || stats.totalFetches < 5) return 1; // Not enough data
  return stats.successfulFetches / stats.totalFetches;
}
```

DAY 6-7: Integration & Testing

Combine with CVE Collector

Create a master collection script that runs both collectors:

```
Master Collector
// collect-all.js
const { fetchCVEs } = require('./collectors/cve-collector');
const { fetchAllFeeds } = require('./collectors/rss-collector');
const { saveToFile, loadFromFile } = require('./lib/storage');

async function collectAll() {
  console.log("=".repeat(50));
  console.log("PITSTACK Data Collection");
  console.log("=".repeat(50));

  const results = {
    timestamp: new Date().toISOString(),
    cves: [],
    news: [],
    errors: []
  };

  // Collect CVEs
  console.log("\n[1/2] Collecting CVEs...");
  try {
    results.cves = await fetchCVEs({ limit: 50 });
    console.log(`  ✓ ${results.cves.length} CVEs collected`);
  } catch (error) {
    results.errors.push({ source: 'CVE', error: error.message });
    console.log(`  ✗ CVE collection failed: ${error.message}`);
  }

  // Collect News
  console.log("\n[2/2] Collecting News...");
  try {
    results.news = await fetchAllFeeds();
    console.log(`  ✓ ${results.news.length} news items collected`);
  } catch (error) {
    results.errors.push({ source: 'News', error: error.message });
    console.log(`  ✗ News collection failed: ${error.message}`);
  }

  // Save combined results
  const allItems = [...results.cves, ...results.news]
    .sort((a, b) => new Date(b.publishedAt) - new Date(a.publishedAt));

  saveToFile('data/feed.json', {
    lastUpdated: results.timestamp,
    itemCount: allItems.length,
    items: allItems
  });

  console.log("\n" + "=".repeat(50));
  console.log(`Total: ${allItems.length} items saved to data/feed.json`);

  return results;
}

collectAll();
```

☐ MODULE: rss-collector.js + collect-all.js

Multi-source RSS collector with parallel fetching, error handling, deduplication, and source reliability tracking. Master script combines all data sources.

Deliverable: collectors/rss-collector.js + collect-all.js + data/feed.json

Time: *~15 hours total*

Week 2 Completion Checklist

- ☐ RSS collector fetches from 5+ sources
- ☐ Parallel fetching with Promise.allSettled
- ☐ Graceful error handling (one failure doesn't crash all)
- ☐ Deduplication removes duplicate articles
- ☐ Data normalized to PITSTACK format
- ☐ collect-all.js runs both CVE and RSS collectors
- ☐ Combined feed saved to data/feed.json

Weeks 3-12: Detailed Roadmap

The following weeks follow the same detailed daily structure. Here's what you'll build:

Week 3: CTF Events & Bug Bounty Collector

- Day 1-2: CTFtime API integration
- Day 3-4: Calculate days until events, difficulty scoring
- Day 5-6: Bug bounty program tracking (scraping/APIs)
- Day 7: Unified collection script for all sources

□ MODULE: ctf-collector.js + bugbounty-collector.js

Event and program tracking with time-based relevance scoring

Deliverable: collectors/*.js

Time: *~12 hours*

Week 4: PostgreSQL Database

- Day 1-2: PostgreSQL installation, SQL fundamentals
- Day 3-4: Schema design for feed_items, sources, users
- Day 5-6: node-postgres integration, parameterized queries
- Day 7: Migrate collectors to use database

□ MODULE: database/schema.sql + database/db.js

Production-ready database with optimized queries

Deliverable: database/*.js + database/*.sql

Time: *~15 hours*

Week 5: Priority Scoring Engine

- Day 1-2: Design scoring algorithm (0-15 scale)
- Day 3-4: Implement scoring for each source type
- Day 5-6: User interest matching, keyword boosting
- Day 7: Recalculation system, testing

□ MODULE: scoring/priority.js

Intelligence ranking system that surfaces what matters

Deliverable: scoring/*.js

Time: *~12 hours*

Week 6: REST API Server

- Day 1-2: Express setup, middleware, routing
- Day 3-4: Feed endpoints (list, filter, search, pagination)
- Day 5-6: Advanced queries, sorting, date ranges
- Day 7: Error handling, input validation, documentation

MODULE: api/server.js + api/routes/*.js

Complete backend API for dashboard consumption

Deliverable: api/**/*.js

Time: *~15 hours*

Week 7: React Dashboard UI

- Day 1-2: Vite + React setup, component basics
- Day 3-4: useState, useEffect, data fetching
- Day 5-6: Tailwind CSS, component library (Card, Badge, Button)
- Day 7: Dashboard layout, feed display, dark mode

MODULE: frontend/src/**

Professional dashboard interface with cybersecurity theme

Deliverable: frontend/**/*.jsx

Time: *~18 hours*

Week 8: Filters, Search & Navigation

- Day 1-2: Filter sidebar (type, severity, date range)
- Day 3-4: Search with debouncing, result highlighting
- Day 5-6: React Router, URL state management
- Day 7: Pagination/infinite scroll, performance

MODULE: FilterSidebar + SearchBar + Routing

Smart filtering system with shareable URLs

Deliverable: frontend/src/components/*.jsx

Time: *~15 hours*

Week 9: User Authentication

- Day 1-2: Auth concepts, bcrypt, JWT
- Day 3-4: Backend auth endpoints (register, login, me)
- Day 5-6: Frontend auth flow, protected routes
- Day 7: User preferences, settings page

MODULE: auth/middleware.js + AuthContext

Complete authentication system with user preferences

Deliverable: auth/*.js + frontend/src/context/*.jsx

Time: *~15 hours*

Week 10: Real-Time Updates

- Day 1-2: WebSocket concepts, Socket.io setup
- Day 3-4: Live feed push, new item notifications
- Day 5-6: Connection handling, reconnection logic
- Day 7: Background jobs with Bull queue

□ MODULE: realtime/socket.js + useSocket hook

Live intelligence feed with automatic updates

Deliverable: realtime/*.js

Time: *~15 hours*

Week 11: Notification System

- Day 1-2: In-app notification center UI
- Day 3-4: Notification storage, read/unread state
- Day 5-6: Real-time notification delivery
- Day 7: Email digests (optional), preferences

□ MODULE: NotificationCenter + notifications/service.js

Alert system for high-priority intelligence

Deliverable: notifications/**

Time: *~12 hours*

Week 12: Deploy & Launch

- Day 1-2: Security hardening, input validation
- Day 3-4: Deployment to Vercel + Railway + Supabase
- Day 5-6: Testing, monitoring, error tracking
- Day 7: Launch preparation, documentation, announcement

□ MODULE: Production PITSTACK

Live, deployed cybersecurity intelligence platform

Deliverable: pitstack.yourdomain.com

Time: *~15 hours*

My Expert Recommendations

After working with many developers learning to build their first real application, here are the patterns I've seen work best:

1. The 30-Minute Rule

When you hit a problem:

- Spend 30 minutes trying to solve it yourself
- Search Google, Stack Overflow, official docs
- If still stuck, ask for a concept explanation (not code)
- Try again with the new understanding
- Only if completely blocked, ask for a code hint—then rewrite it yourself

MY RECOMMENDATION

The struggle IS the learning. When you fight through a bug for an hour, you remember the solution forever. When someone gives you the answer, you forget it in a day.

2. Commit Early, Commit Often

Git commits are not just for backup—they're your safety net:

- Commit after each working feature (even small ones)
- Write descriptive commit messages
- If you break something, you can always go back
- Aim for 3-5 commits per day of coding

```
Good Commit Messages
git commit -m "Add CVE severity filtering"
git commit -m "Fix: handle missing CVSS scores gracefully"
git commit -m "Refactor: extract RSS parsing to separate module"
```

3. Read Error Messages Carefully

Beginners often panic at errors. Professionals read them carefully:

- The error message usually tells you exactly what's wrong
- Look for the file name and line number
- Read the stack trace from bottom to top
- Google the exact error message (in quotes)

4. Don't Over-Engineer Early

Build the simple version first, then improve:

- Get it working, then make it elegant

- Don't add features you might need 'someday'
- Refactor when you feel pain, not before
- Perfect is the enemy of done

5. Take Breaks

Your brain needs time to process new information:

- Take a 5-minute break every 25-30 minutes (Pomodoro technique)
- Step away when frustrated—solutions often come during breaks
- Sleep on difficult problems
- Exercise improves coding ability (seriously)

What Success Looks Like

After completing this 12-week program, you will have:

A Complete Product

- Live PITSTACK deployment at your own domain
- CVE, news, CTF, and bug bounty intelligence aggregation
- Priority scoring that surfaces what matters
- Real-time updates via WebSocket
- User authentication and preferences
- Professional, responsive dashboard

Real Skills

- JavaScript/Node.js proficiency
- React and modern frontend development
- PostgreSQL database design
- REST API architecture
- Authentication and security
- Real-time systems
- Deployment and DevOps

Professional Practices

- Git workflow and version control
- Code organization and modularity
- Error handling and resilience
- Documentation
- Testing mindset

Start Today

Week 1, Day 1: Set up your environment and write your first CVE fetch.

Don't wait until you 'feel ready.' Start now.

Every expert was once a beginner who decided to start.