Web Reaper: Harvesting Digital Footprints

Project Overview

Web Reaper is an advanced web crawler meticulously crafted in C++ for the purpose of efficient data discovery and analysis, specifically within the realm of cybersecurity. The project is spearheaded by the team "Web Reapers," under the leadership of Momna Jamil, and comprises the following members:

- Momna Jamil - 2023336 (Developer)
- Rida Shahid Malik - 2023610 (Tester)
- Muhammad Sofia Faisal - 2023519 (Documentation)

The primary objective of this endeavor is to enhance threat intelligence gathering and facilitate vulnerability assessments through the automation of data collection from diverse online sources.

Use Cases in Cybersecurity

The utility of the Web Reaper extends across several critical domains within cybersecurity:

1. Threat Intelligence Collection: The web crawler is designed to continuously monitor and scrape data from forums, social media platforms, and news websites, thereby identifying emerging threats and vulnerabilities. This aggregation of information equips cybersecurity professionals with timely insights into potential risks and attack vectors.

2. Vulnerability Discovery: By targeting specific websites, the crawler can effectively identify outdated software versions, misconfigurations, or exposed sensitive information that may be exploited by malicious actors. This proactive approach empowers organizations to remediate vulnerabilities before they can be leveraged in attacks.

3. Incident Response Support: In the event of a cybersecurity incident, swift access to pertinent information is paramount. The crawler facilitates rapid data gathering regarding affected systems or compromised accounts from multiple sources, thereby aiding incident response teams in their investigations.

4. Compliance Monitoring: The tool assists organizations in ensuring adherence to regulatory standards by scraping data related to best practices and security guidelines from authoritative sources. This capability enables organizations to remain informed about the latest requirements and adjust their security measures accordingly.

5. Competitive Security Analysis: Organizations can leverage the web crawler to monitor competitors or industry peers for security breaches or data leaks. This competitive

intelligence informs an organization's own security strategies, enhancing defenses against similar threats.

Through the implementation of this web crawler, we aspire to provide cybersecurity professionals with a formidable tool for proactive threat detection and vulnerability management, ultimately bolstering the overall security posture of organizations.

Project Objectives

The objectives guiding this project are as follows:

1. Efficiently process and store discovered data.
2. Dynamically prioritize and manage crawling tasks based on response times.
3. Rapidly detect and flag anomalies in web responses in real-time.
4. Facilitate comprehensive data analysis for further research or application development.

Problem Statement and Motivation

Problem Statement: Current web crawlers frequently encounter challenges in efficiently managing substantial volumes of data while ensuring timely responses during high-traffic conditions. This inefficiency can lead to missed opportunities for critical data collection.

Motivation: By developing a highly efficient web crawler, we aim to enhance data collection processes across various applications, thereby improving accessibility to vital information and enabling superior decision-making based on real-time data analysis.

Detailed Project Plan

Scope and Requirements: This project will focus on backend processing of web data for discovery and analysis, with optional graphical user interface (GUI) development. The primary emphasis will be on efficient data handling and analytical capabilities.

Data Structures Utilized: The implementation employs various data structures that enhance efficiency in managing URLs and crawling tasks:

1. Queues:
   - Purpose: Queues are employed to manage URLs that require crawling, adhering to the First-In-First-Out (FIFO) principle.
   - Role in Project: Utilizing a queue allows the web crawler to systematically explore web pages while maintaining a clear order of operations. As URLs are fetched and processed, they are removed from the front of the queue, while new URLs discovered during crawling can be appended to the rear.

2. Maps:

- Purpose: Maps (or hash tables) are utilized to track discovered pages and their statuses.
- Role in Project: The implementation of a map enables efficient storage and retrieval of page information, including response times and visitation status. This prevents redundant requests to previously crawled URLs, optimizing resource utilization during the crawling process.

3. Linked Lists:
- Purpose: A linked list is employed to store lists of discovered pages alongside their response times.
- Role in Project: In this web crawler, a linked list facilitates easy insertion and deletion of nodes representing discovered pages. As pages are crawled and their response times recorded, new nodes can be added without necessitating resizing an array or vector—an especially advantageous feature when dealing with large datasets or numerous links on a single page.

Algorithms Implemented: The following algorithms are integral to the functionality of Web Reaper:

1. Socket Connection Establishment: Establishes connections using system calls while adeptly handling connection errors.
2. HTTP GET Request Construction: Constructs valid HTTP GET request strings formatted according to HTTP/1.1 specifications.
3. Response Handling: Receives server data in chunks until no further data is available, ensuring comprehensive capture of all response content.
4. URL Extraction: Parses HTTP responses to extract URLs using predefined patterns (e.g., identifying href attributes).
5. URL Validation: Verifies extracted URLs against criteria such as permitted domains and file types to filter out undesirable links.
6. Response Time Calculation: Measures request-response times for performance analysis.
7. Statistics Calculation: Maintains statistics such as the number of pages discovered, failed requests, and response times for reporting purposes.

Code Structure

The codebase is organized into several key sections that encapsulate the functionality of the web crawler:

1. Header Files and Namespaces

```
#include "clientSocket.h"
#include "parser.h"
#include <iostream>
#include <fstream>
```

```
#include <thread>
#include <mutex>
#include <map>
#include <iomanip>
#include <condition_variable>

using namespace std;
```

This section includes necessary header files for socket communication (clientSocket.h), HTML parsing (parser.h), as well as standard libraries for input/output operations, file handling, multithreading, synchronization mechanisms, etc.

2. Data Structures

SiteNode and SiteQueue

```
struct SiteNode {
  string hostname;
  int depth;
  SiteNode* next;

  SiteNode(string host, int d) : hostname(host), depth(d), next(nullptr) {}
};

class SiteQueue {
private:
  SiteNode* head;
  SiteNode* tail;

public:
  SiteQueue() : head(nullptr), tail(nullptr) {}

  ~SiteQueue() {
    clear();
  }

  void push(string hostname, int depth) {
    SiteNode* newNode = new SiteNode(hostname, depth);
    if (!head) {
      head = tail = newNode;
    } else {
      tail->next = newNode;
      tail = newNode;
    }
```

```
    }

    pair<string, int> front() {
        if (!head) throw runtime_error("Queue is empty.");
        return {head->hostname, head->depth};
    }

    void pop() {
        if (!head) return;
        SiteNode* temp = head;
        head = head->next;
        delete temp;
        if (!head) tail = nullptr;
    }

    bool empty() const {
        return head == nullptr;
    }

    void clear() {
        while (!empty()) pop();
    }
};
```

SiteNode: This structure encapsulates a node within a linked list that comprises a hostname alongside its associated depth within the crawl hierarchy.
SiteQueue: This class implements a linked list structure designed for efficiently managing SiteNode objects. It provides methods for adding nodes (push), retrieving nodes (front), and removing nodes (pop) while preserving order based on insertion sequence.

UrlNode and UrlList

```
struct UrlNode {
    string url;
    UrlNode* next;

    UrlNode(string u) : url(u), next(nullptr) {}
};

class UrlList {
private:
    UrlNode* head;

public:
```

```
  UrlList() : head(nullptr) {}

  ~UrlList() {
    clear();
  }

  void add(string url) {
    UrlNode* newNode = new UrlNode(url);
    if (!head) {
      head = newNode;
    } else {
      UrlNode* temp = head;
      while (temp->next) temp = temp->next;
      temp->next = newNode;
    }
  }

  UrlNode* getHead() const {
    return head;
  }

  void clear() {
    while (head) {
      UrlNode* temp = head;
      head = head->next;
      delete temp;
    }
  }
};
```

UrlNode: Represents an individual URL node within a linked list structure.
UrlList: This class manages a linked list containing starting URLs designated for crawling purposes. It facilitates dynamic addition of URLs without necessitating pre-allocation of space.

3. Configuration Structure

```
typedef struct {
  int crawlDelay = 1000; // Delay between requests in milliseconds
  int maxThreads = 10; // Maximum number of concurrent threads
  int depthLimit = 10; // Maximum crawl depth
  int pagesLimit = 10; // Maximum number of pages to crawl per site
  int linkedSitesLimit = 10; // Maximum number of linked sites to discover
  UrlList startUrls; // List of starting URLs for crawling
```

```
} Config;

Config readConfigFile();
```

The Config structure encapsulates various settings pertinent to the crawler's operation—such as delays between requests and limitations on threads or pages crawled.
The function readConfigFile is responsible for reading configuration settings from an external file named config.txt, thereby allowing dynamic adjustment of parameters without recompilation.

4. Crawler State Management

```
struct CrawlerState {
   int threadsCount; // Count of active threads
   SiteQueue pendingSites; // Queue for pending sites to crawl
   map<string, bool> discoveredSites; // Map to track discovered sites
};
```

The CrawlerState structure maintains critical state information regarding active threads engaged in crawling activities as well as tracking pending sites within the queue alongside previously discovered sites—thereby preventing redundant processing.

5. Main Functionality

Main Entry Point

```
int main(int argc, const char* argv[]) {
   config = readConfigFile();
   initialize();
   scheduleCrawlers();
   return 0;
}
```

The main function serves as an entry point for execution by reading configuration settings from a file, initializing state variables for crawling tasks, and scheduling those tasks accordingly.

Initialization Function

```
void initialize() {
  crawlerState.threadsCount = 0;

  UrlNode* urlNode = config.startUrls.getHead();
  while (urlNode) {
```

```
    string url = urlNode->url;
    crawlerState.pendingSites.push(getHostnameFromUrl(url), 0);
    crawlerState.discoveredSites[getHostnameFromUrl(url)] = true;
    urlNode = urlNode->next;
  }
}
```

This function establishes initial conditions by populating pendingSites with starting URLs derived from configuration settings while concurrently marking them as already discovered—thereby preventing unnecessary reprocessing during subsequent crawls.

Scheduling Crawlers

```
void scheduleCrawlers() {
  while (crawlerState.threadsCount != 0 || !crawlerState.pendingSites.empty()) {
    m_mutex.lock();
    threadFinished = false;

    while (!crawlerState.pendingSites.empty() && crawlerState.threadsCount <
config.maxThreads) {
      auto nextSite = crawlerState.pendingSites.front();
      crawlerState.pendingSites.pop();
      crawlerState.threadsCount++;

      thread t(startCrawler, nextSite.first, nextSite.second, ref(crawlerState));
      if (t.joinable()) t.detach();
    }
    m_mutex.unlock();

    unique_lock<mutex> m_lock(m_mutex);
    while (!threadFinished) m_condVar.wait(m_lock);
  }
}
```

The scheduleCrawlers function orchestrates multithreading by dynamically creating new threads dedicated to executing crawling tasks as long as there are pending sites available within pendingSites, ensuring that active threads remain below pre-defined limits established in configuration settings.

Crawler Execution Function

```
void startCrawler(string hostname, int currentDepth, CrawlerState& crawlerState) {
  ClientSocket clientSocket(hostname, 80, config.pagesLimit, config.crawlDelay);
  SiteStats stats = clientSocket.startDiscovering();
```

```
  m_mutex.lock();
  cout << "--------------------------------------------------------------------------------" << endl;
  cout << "Website: " << stats.hostname << endl;
  cout << "Depth (distance from starting pages): " << currentDepth << endl;
  cout << "Number of Pages Discovered: " << stats.discoveredPages.size() << endl;
  cout << "Number of Pages Failed to Discover: " << stats.numberOfPagesFailed << endl;
  cout << "Number of Linked Sites: " << stats.linkedSites.size() << endl;

  if (currentDepth < config.depthLimit) {
    for (int i = 0; i < min(int(stats.linkedSites.size()), config.linkedSitesLimit); i++) {
      string site = stats.linkedSites[i];
      if (!crawlerState.discoveredSites[site]) {
        crawlerState.pendingSites.push(site, currentDepth + 1);
        crawlerState.discoveredSites[site] = true;
      }
    }
  }
  crawlerState.threadsCount--;
  threadFinished = true;
  m_mutex.unlock();

  m_condVar.notify_one();
}
```

Each thread executes startCrawler which establishes a socket connection via ClientSocket retrieves site statistics through startDiscovering prints relevant statistics about the crawling process including discovered links at each depth level while adding newly found links into pendingSites. This method ensures that only unvisited links are queued for further exploration—effectively managing crawl depth dynamically based on configuration parameters.

Parsing Functions

The parsing functions are defined in parser.h which includes methods for extracting URLs from HTTP responses:

```
string getHostnameFromUrl(string url);
string getHostPathFromUrl(string url);
LinkedList extractUrls(string httpText);
bool verifyUrl(string url);
bool verifyType(string url);
bool verifyDomain(string url);
bool hasSuffix(string str, string suffix);
```

string reformatHttpResponse(string text);

These functions serve various purposes:

- getHostnameFromUrl: Extracts hostname from given URL by parsing its structure.
- getHostPathFromUrl: Extracts path component from given URL utilizing string manipulation techniques.
- extractUrls: Parses HTTP response text using regular expressions or predefined patterns to identify valid URLs embedded within HTML content.
- verifyUrl: Validates extracted URLs against predefined criteria such as allowed domains or file types—ensuring only relevant links are processed further.
- verifyType: Checks whether extracted URLs conform with allowed types (e.g., excluding CSS or JS files).
- verifyDomain: Ensures extracted domains comply with specified criteria—such as being part of recognized top-level domains.
- hasSuffix: Utility function that checks if string ends with specific suffixes—facilitating domain validation checks.
- reformatHttpResponse: Cleans up raw HTTP response data by removing unwanted characters—enhancing parsing accuracy during URL extraction processes.

Error Handling

Error handling is integrated throughout various parts of the codebase to ensure robustness against unexpected conditions:

if (!head) throw runtime_error("Queue is empty.");

This line ensures that operations on an empty queue do not lead to undefined behavior—thereby enhancing stability during execution under adverse conditions.

ClientSocket Header File (clientSocket.h)
This header file defines the ClientSocket class responsible for establishing connections with specific websites and managing HTTP requests.

- SiteStats Structure
    - Contains fields for tracking hostname, response time statistics (average, minimum, maximum), number of failed page discoveries, linked sites found during crawling, and a list of discovered pages along with their response times.
- ClientSocket Class
    - Methods include:
        - ClientSocket(string hostname, int port=80, int pagesLimit=-1, int crawlDelay=1000): Constructor initializes socket parameters.
        - SiteStats startDiscovering(): Orchestrates website discovery by managing pending pages and extracting discovered URLs.

Parser Header File (parser.h)
This header file declares functions essential for parsing URLs from raw HTTP responses.

- Functions include:
    - getHostnameFromUrl(string url): Extracts hostname from a URL.
    - getHostPathFromUrl(string url): Extracts path component from a URL.
    - extractUrls(string httpRaw): Parses raw HTTP text for valid URLs.
    - Validation functions ensure URL integrity against specified criteria.

Parser Implementation File (parser.cpp)
This section implements parsing functionality declared in parser.h.

- Key functions include:
    - extractUrls: Processes raw HTML text using predefined patterns (e.g., "href=") to locate potential links while verifying their validity before adding them to a list.
    - Validation functions check extracted URLs against allowed domains or file types.

Main Program (main.cpp)
The main program initializes configurations, manages threads for crawling tasks, and schedules those tasks effectively.

- Key components:
    - readConfigFile(): Reads settings from an external configuration file.
    - initialize(): Sets up initial state by populating pending sites with starting URLs derived from configuration settings.
    - scheduleCrawlers(): Manages multithreading by creating new threads dedicated to executing crawling tasks based on available pending sites.

ClientSocket Implementation File (clientSocket.cpp)
This implementation file contains methods defined in the ClientSocket class.

- Key functionalities:
    - startConnection(): Establishes a socket connection using system calls; handles errors gracefully.
    - createHttpRequest(): Constructs an HTTP GET request string formatted according to HTTP/1.1 specifications.
    - startDiscovering(): Manages crawling by sending requests, receiving responses, extracting URLs from responses while maintaining performance metrics.

Highlights of Incomplete Parts

1. Error Handling Enhancements
    - There is a need for more robust error handling mechanisms that provide detailed feedback on connection issues or failed requests.
2. Testing Framework
    - A formal testing framework has not yet been established; unit tests need integration for each component of the crawler to ensure reliability and correctness.
3. Performance Optimization

    - Opportunities exist for optimizing performance regarding thread management and network request handling; future work will focus on refining these processes for improved efficiency.
4. Documentation
    - Comprehensive documentation for each function and class is still pending; this will be essential for future maintenance and onboarding new developers.

Challenges Faced During Development

1. Managing Concurrent Threads
    - The introduction of multiple threads led to synchronization issues; mutexes and condition variables were implemented to ensure safe access to shared resources without race conditions.
2. URL Extraction Accuracy
    - Extracting valid URLs proved challenging due to varying formats; rigorous validation checks were implemented to filter out invalid links effectively.
3. Connection Timeouts
    - Frequent timeouts when connecting to certain websites were encountered; connection handling logic was adjusted with retries using exponential backoff strategies.
4. Handling Large Volumes of Data
    - Memory management became a concern as more pages were processed; efficient data structures such as queues and maps were utilized along with cleanup routines for unused resources.

Conclusion

The Web Reaper project embodies an innovative approach to web crawling tailored specifically for cybersecurity applications. By implementing efficient data collection processes alongside robust algorithms within a modular architecture this web crawler aims not only to enhance threat detection capabilities but also facilitate effective vulnerability management for cybersecurity professionals. The structured design ensures that real-time data analysis is both feasible and effective—positioning Web Reaper as an invaluable asset in fortifying organizational defenses against emerging threats in an increasingly complex digital landscape.