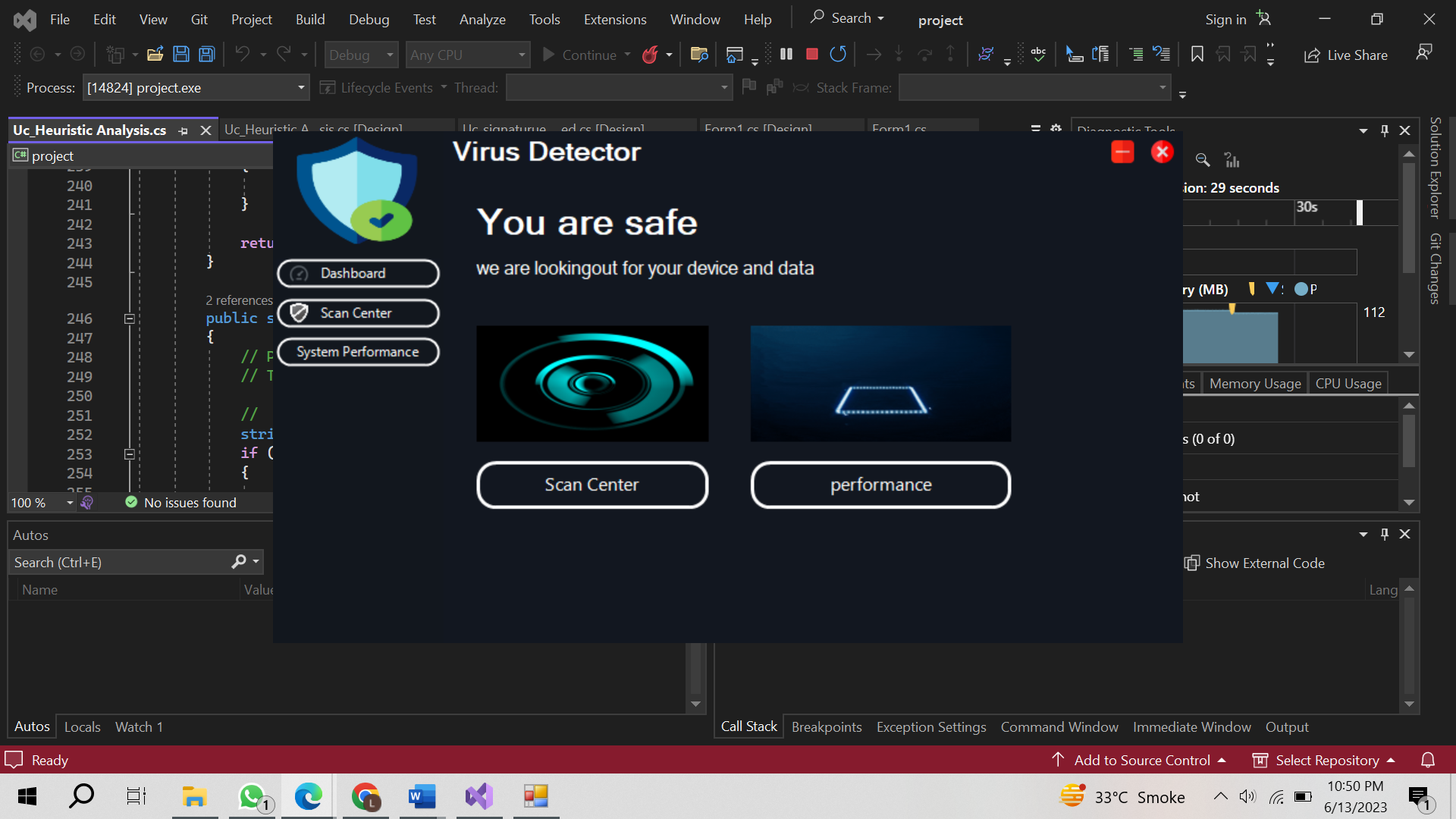
**Virus Detector**

## Introduction:

We have developed a powerful 'Virus Detector' software that combines heuristic analysis and signature-based analysis to provide comprehensive and effective threat detection. By leveraging intelligent algorithms and behavioral analysis, the software identifies suspicious patterns and behaviors that indicate the presence of malware or viruses. Additionally, it utilizes a robust signature database to detect known threats accurately. With automatic updates and customizable scanning options, our 'Virus Detector' offers users reliable protection, ensuring the security of their systems against both known and emerging threats.



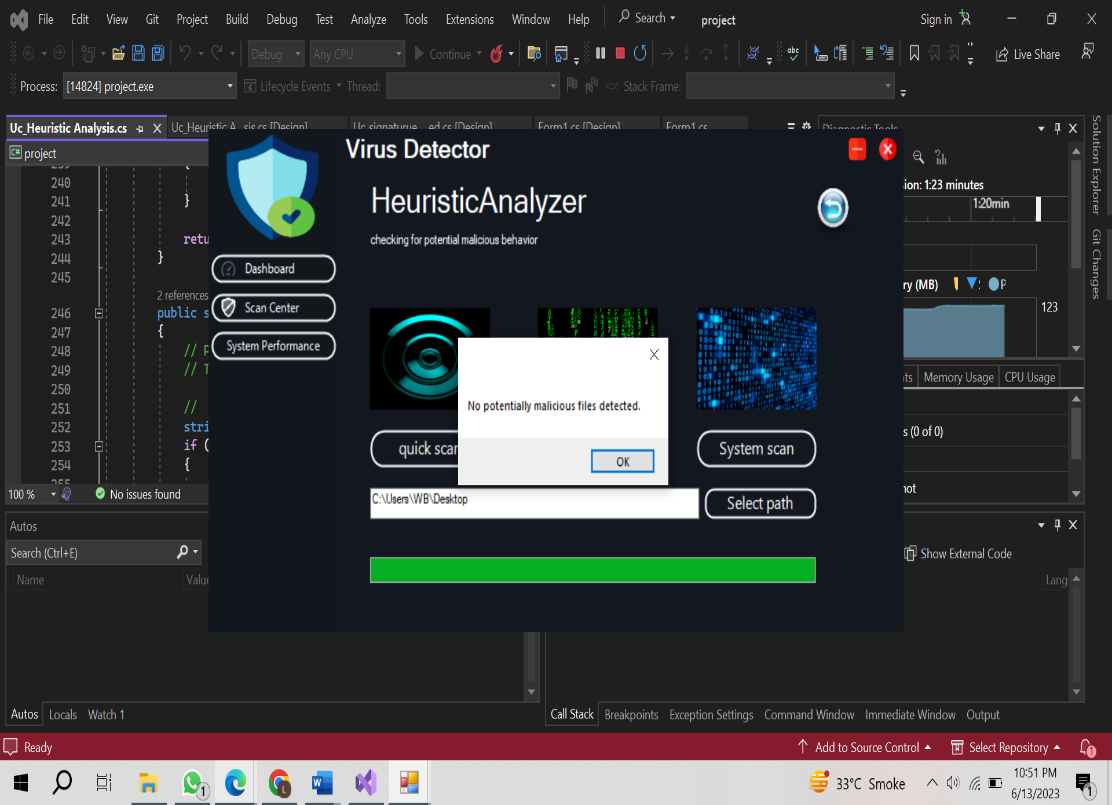
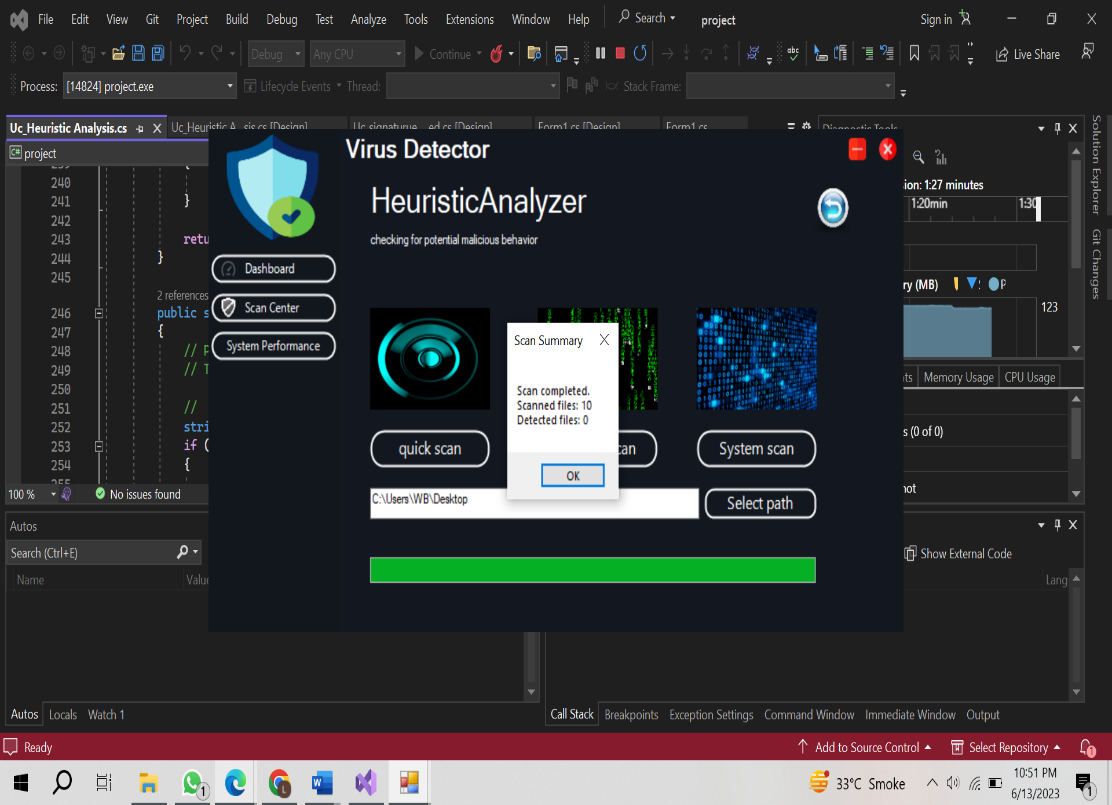
Scan Center:

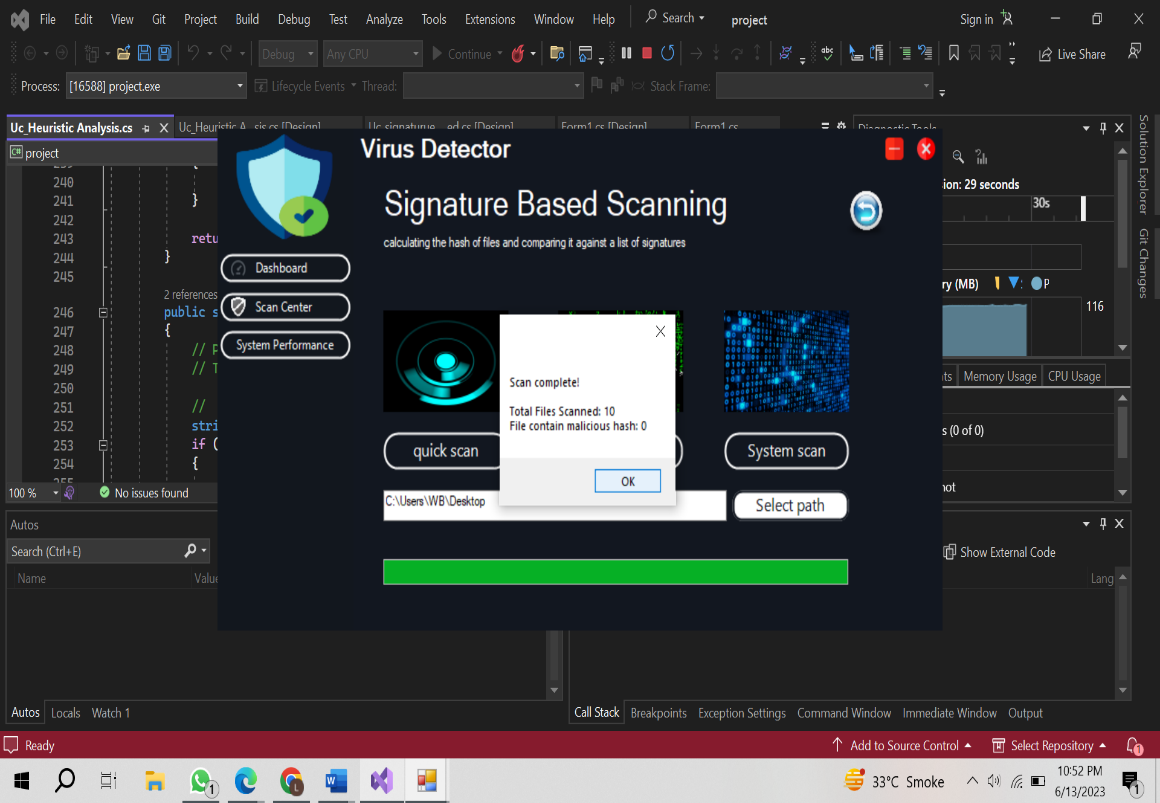


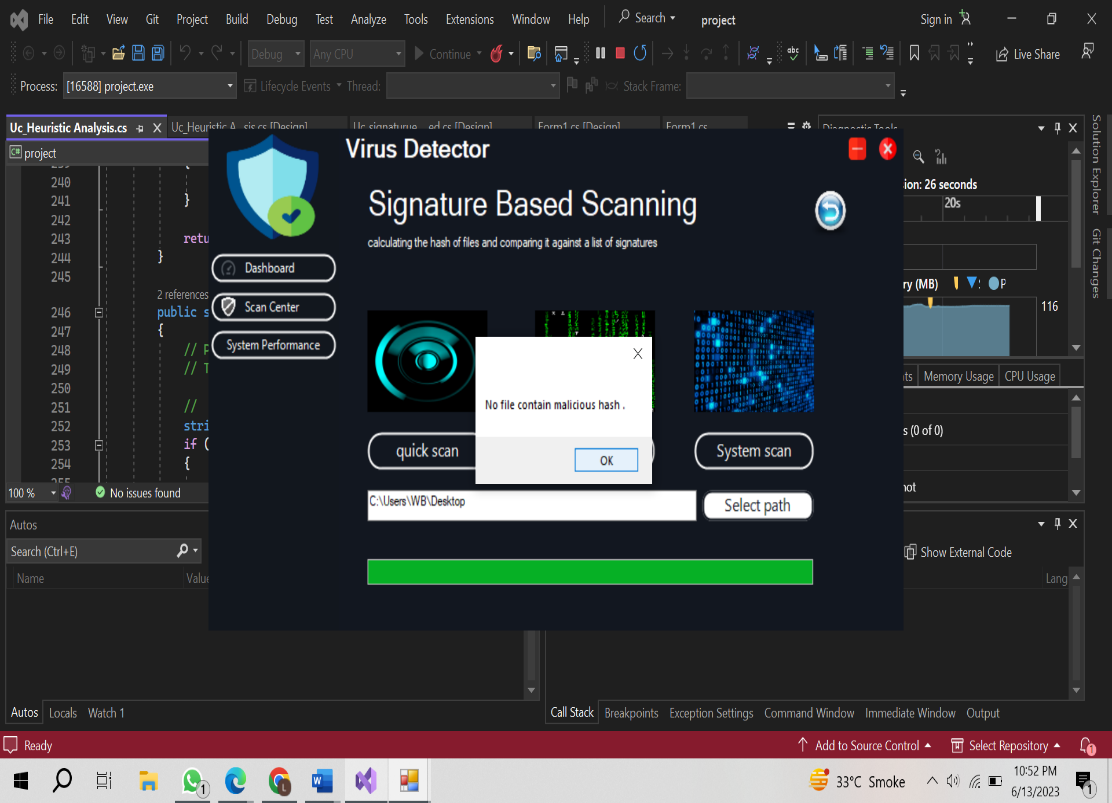
## Types of Scanning:

Quick, custom, and system scans are different types of scanning. Here's a brief description of each:

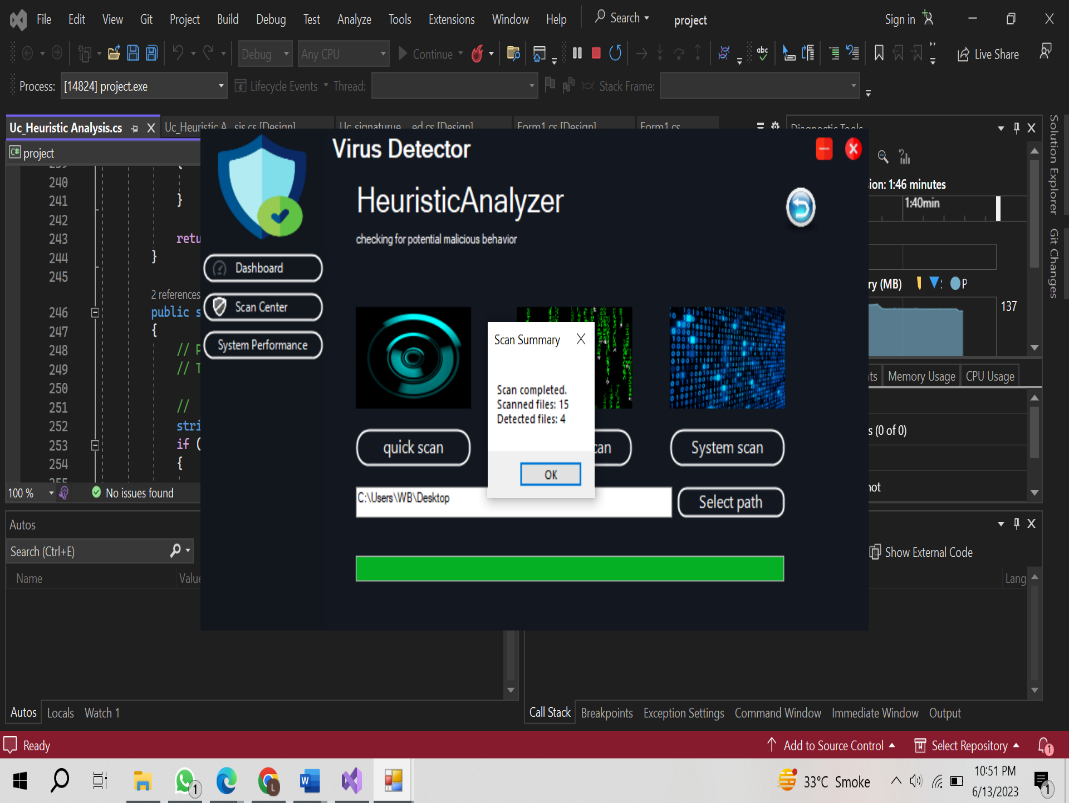
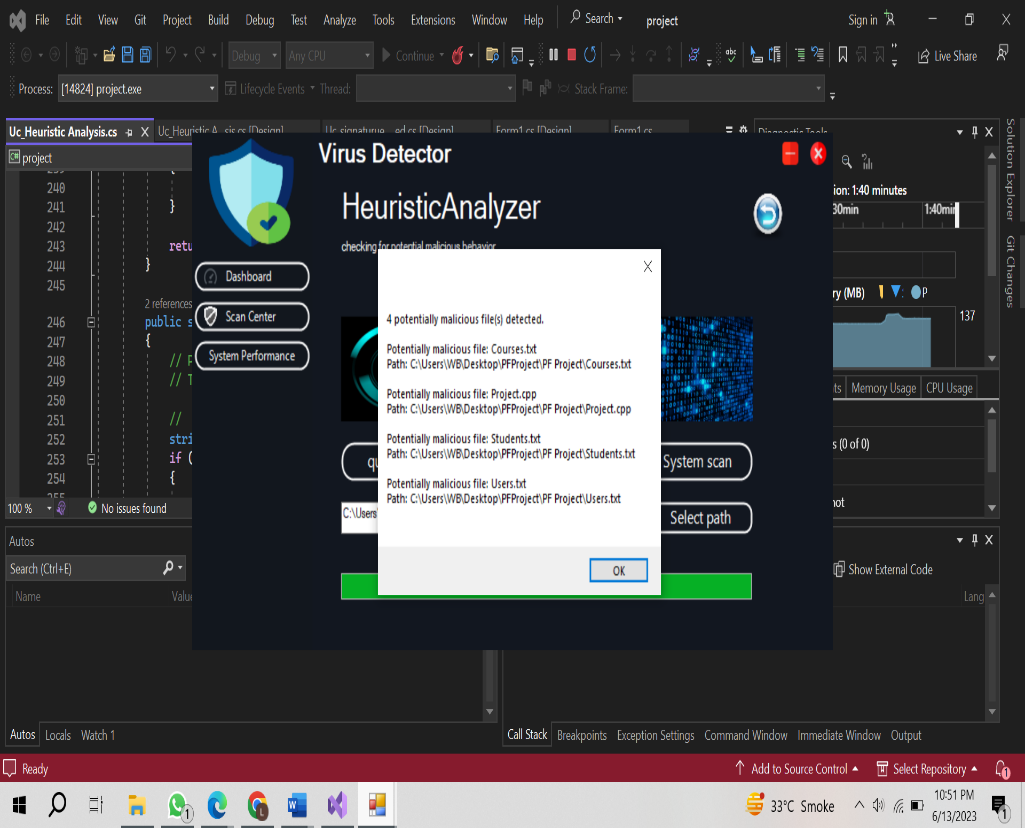
1. Quick Scan: A quick scan is a fast and targeted scan that focuses on critical areas of the system where malware is most likely to reside or be actively executed. It typically scans common infection points such as running processes, system files, and areas of the operating system that are commonly targeted by malware. A quick scan is designed to provide a rapid assessment of potential threats, allowing users to quickly identify and address any immediate security risks.

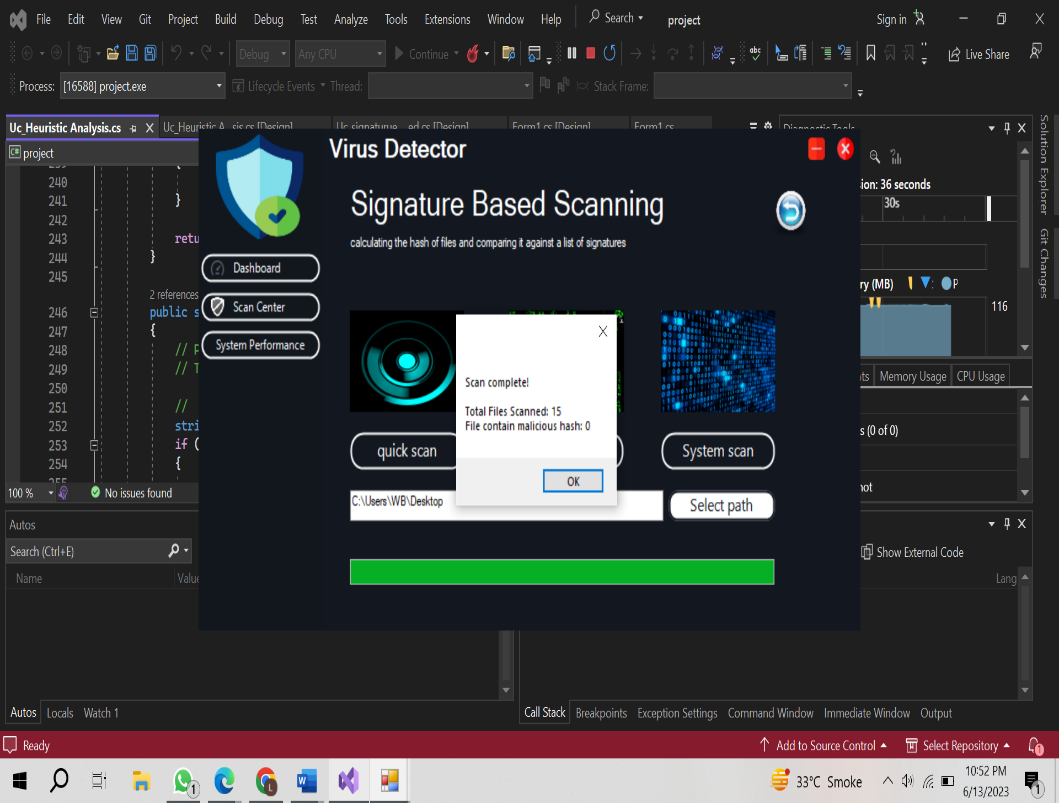
**Quick Scan using Heuristic Analyzeer**

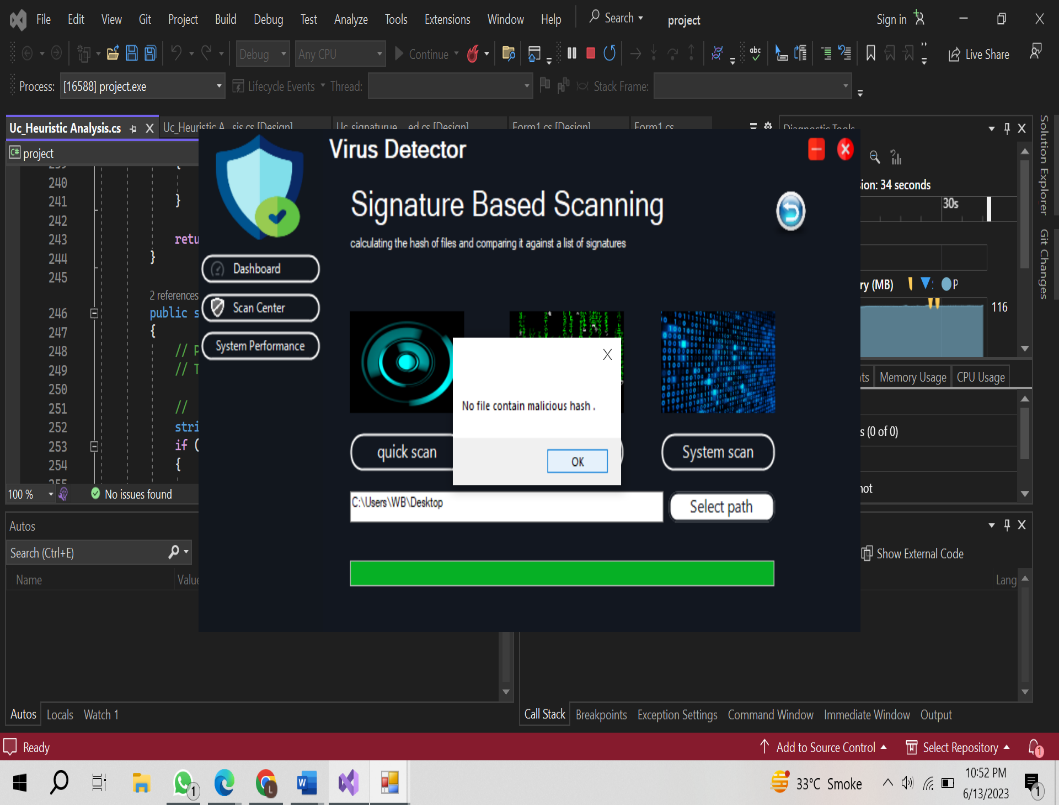
**Quick Scan using Signature-based Scanning**

****

1. Custom Scan: A custom scan provides users with the flexibility to choose specific files, folders, or drives to be scanned according to their preferences. It allows users to target specific areas of their system that they suspect may contain malware or to scan specific files they deem suspicious. Custom scans are useful for performing more in-depth scans on specific areas or files of concern, providing a tailored approach to security based on user-defined parameters.

**Custom Scan using Heuristic Analyzer**

**Custom Scan using Signature-based Analyzer**



1. System Scan: A system scan, also known as a full scan or comprehensive scan, thoroughly examines all files and directories on the computer, including the operating system, installed applications, system files, and user data. It scans the entire system to detect any potential malware or malicious code. A system scan is a more extensive and time-consuming process compared to quick scans or custom scans, as it covers the entire system, ensuring a more comprehensive assessment of potential threats.

Two methods of scanning are performed.

1. Heuristic analysis is often employed to identify potential issues, vulnerabilities, or patterns in software systems. This method involves analyzing the code, system behavior, or user interactions to detect possible errors, security flaws, or usability problems. Heuristic analysis is particularly useful when formal verification methods or exhaustive testing are impractical or time-consuming.

### Code:

using project.Dashboard;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace project.scancenter

{

public partial class Uc\_Heuristic\_Analysis : UserControl

{

private HeuristicAnalyzer heuristicAnalyzer;

public Uc\_Heuristic\_Analysis()

{

InitializeComponent();

heuristicAnalyzer = new HeuristicAnalyzer();

}

private void Uc\_Heuristic\_Analysis\_Load(object sender, EventArgs e)

{

}

private void progressBarscan\_Click(object sender, EventArgs e)

{

// Progress bar

}

private async void Btn\_Quickscan\_Click(object sender, EventArgs e)

{

// Quick scan

string selectedPath = textBox1.Text;

if (!string.IsNullOrEmpty(selectedPath))

{

await PerformQuickScan(selectedPath);

}

else

{

MessageBox.Show("Please select a folder to scan.");

}

}

private async void Btn\_Custom\_scan\_Click\_1(object sender, EventArgs e)

{

// Custom scan

string selectedPath = textBox1.Text;

if (!string.IsNullOrEmpty(selectedPath))

{

await PerformCustomScan(selectedPath);

}

else

{

MessageBox.Show("Please select a folder to scan.");

}

}

private async void Btn\_system\_scan\_Click\_1(object sender, EventArgs e)

{

// System scan

string systemPath = Environment.GetFolderPath(Environment.SpecialFolder.Windows);

await PerformSystemScan(systemPath);

}

private void BtnSelect\_Click\_1(object sender, EventArgs e)

{

}

private async Task PerformQuickScan(string path)

{

await Task.Run(() =>

{

ScanFiles(path, SearchOption.TopDirectoryOnly);

});

}

private async Task PerformCustomScan(string path)

{

await Task.Run(() =>

{

ScanFiles(path, SearchOption.AllDirectories);

});

}

private async Task PerformSystemScan(string path)

{

await Task.Run(() =>

{

ScanFiles(path, SearchOption.AllDirectories);

});

}

private void ScanFiles(string path, SearchOption searchOption)

{

try

{

string[] files = Directory.GetFiles(path, "\*.\*", searchOption);

int totalFiles = files.Length;

int scannedFiles = 0;

int detectedFiles = 0;

StringBuilder detectedFilesInfo = new StringBuilder();

foreach (string file in files)

{

// Update progress bar

UpdateProgressBar(scannedFiles, totalFiles);

string filePath = file; // Initialize the filePath variable

if (HeuristicAnalyzer.IsMalicious(filePath)) // Specify HeuristicAnalyzer class

{

// File is flagged as potentially malicious

detectedFiles++;

detectedFilesInfo.AppendLine($"Potentially malicious file: {Path.GetFileName(filePath)}");

detectedFilesInfo.AppendLine($"Path: {filePath}");

detectedFilesInfo.AppendLine();

}

// Increase the count of scanned files regardless of the detection

scannedFiles++;

}

// Update progress bar after scanning is complete

UpdateProgressBar(scannedFiles, totalFiles);

// Show detection result after scanning completes

if (detectedFiles > 0)

{

MessageBox.Show($"{detectedFiles} potentially malicious file(s) detected.\n\n{detectedFilesInfo.ToString()}");

}

else

{

MessageBox.Show("No potentially malicious files detected.");

}

// Display scan summary

DisplayScanSummary(scannedFiles, detectedFiles);

}

catch (Exception ex)

{

MessageBox.Show($"An error occurred during the scan: {ex.Message}");

}

}

private bool HasCodeInjection(string filePath)

{

// code of injection detection here

// This can involve analyzing the file's content for suspicious code patterns or known code injection techniques

// Check if the file contains suspicious code injection patterns

string fileContent = File.ReadAllText(filePath);

if (HeuristicAnalyzer.HasCodeInjection(fileContent)) // Specify HeuristicAnalyzer class

{

return true; // Code injection detected

}

return false; // No code injection detected

}

private bool HasUnauthorizedModifications(string filePath)

{

// unauthorized modifications detection

// This can involve comparing the file's metadata, checksum, or digital signatures against trusted references

// Check if the file's metadata has been tampered with

FileInfo fileInfo = new FileInfo(filePath);

DateTime lastWriteTime = fileInfo.LastWriteTime;

DateTime creationTime = fileInfo.CreationTime;

// Compare the last write time with the creation time

if (HeuristicAnalyzer.HasUnauthorizedModifications(filePath)) // Specify HeuristicAnalyzer class

{

// Unauthorized modifications detected

return true;

}

return false; // No unauthorized modifications detected

}

private void UpdateProgressBar(int scannedFiles, int totalFiles)

{

// Calculate progress percentage

int progressPercentage = (scannedFiles \* 100) / totalFiles;

// Update progress bar value

this.Invoke((MethodInvoker)delegate

{

progressBarscan.Value = progressPercentage;

});

}

private void DisplayScanSummary(int scannedFiles, int detectedFiles)

{

string scanSummary = $"Scan completed.\nScanned files: {scannedFiles}\nDetected files: {detectedFiles}";

this.Invoke((MethodInvoker)delegate

{

MessageBox.Show(scanSummary, "Scan Summary");

});

}

private void BtnBack\_Click\_1(object sender, EventArgs e)

{

UC\_scan uC\_Scan = new UC\_scan();

uC\_Scan.Show();

this.Hide();

}

private void BtnSelect\_Click(object sender, EventArgs e)

{

if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)

{

textBox1.Text = folderBrowserDialog1.SelectedPath.ToString();

}

else

{

MessageBox.Show("Please choose a folder to scan.");

}

}

}

public class HeuristicAnalyzer

{

public static bool IsMalicious(string filePath)

{

// Check for suspicious behavior and characteristics in the file

if (HasCodeInjection(filePath) || HasUnauthorizedModifications(filePath))

{

return true; // File is flagged as potentially malicious

}

return false; // File is clean

}

public static bool HasCodeInjection(string filePath)

{

// Perform code injection detection logic here

// This can involve analyzing the file's content for suspicious code patterns or known code injection techniques

// Check if the file contains suspicious code injection patterns

string fileContent = File.ReadAllText(filePath);

if (fileContent.Contains("eval(base64\_decode"))

{

return true; // Code injection detected

}

return false; // No code injection detected

}

public static bool HasUnauthorizedModifications(string filePath)

{

// Perform unauthorized modifications detection logic here

// This can involve comparing the file's metadata, checksum, or digital signatures against trusted references

// Check if the file's metadata has been tampered with

FileInfo fileInfo = new FileInfo(filePath);

DateTime lastWriteTime = fileInfo.LastWriteTime;

DateTime creationTime = fileInfo.CreationTime;

// Compare the last write time with the creation time

if (lastWriteTime < creationTime)

{

return true; // Unauthorized modifications detected

}

return false; // No unauthorized modifications detected

}

}

}

1. Signature-based analysis : is a method commonly used in cybersecurity to detect and identify known threats, such as malware, viruses, or malicious code. It involves comparing digital signatures or patterns derived from known malicious entities against files or network traffic to identify matching signatures and indicate the presence of a threat.

### Code:

using System;

using System.IO;

using System.Security.Cryptography;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace project.scancenter

{

public partial class Uc\_signaturue\_based : UserControl

{

private SignatureBasedDetection signatureBasedDetection;

private string signaturesFilePath = @"C:\Program Files\signature.txt";

public Uc\_signaturue\_based()

{

InitializeComponent();

signatureBasedDetection = new SignatureBasedDetection(signaturesFilePath);

}

private void BtnBack\_Click(object sender, EventArgs e)

{

UC\_scan uC\_Scan = new UC\_scan();

uC\_Scan.Show();

this.Hide();

}

private async void Btn\_system\_scan\_Click(object sender, EventArgs e)

{

// Perform a system scan by passing the system path

string systemPath = Environment.GetFolderPath(Environment.SpecialFolder.Windows);

await PerformSystemScan(systemPath);

}

private async void Btn\_Custom\_scan\_Click(object sender, EventArgs e)

{

// Perform a custom scan by passing the selected path

string selectedPath = textBox1.Text;

if (!string.IsNullOrEmpty(selectedPath))

{

await PerformCustomScan(selectedPath);

}

else

{

MessageBox.Show("Please select a folder to scan.");

}

}

private async void Btn\_Quickscan\_Click(object sender, EventArgs e)

{

// Perform a quick scan by passing the selected path

string selectedPath = textBox1.Text;

if (!string.IsNullOrEmpty(selectedPath))

{

await PerformQuickScan(selectedPath);

}

else

{

MessageBox.Show("Please select a folder to scan.");

}

}

private async Task PerformSystemScan(string path)

{

await Task.Run(() =>

{

// Perform the scan by calling the ScanFiles method

ScanFiles(path, SearchOption.AllDirectories);

});

}

private async Task PerformCustomScan(string path)

{

await Task.Run(() =>

{

// Perform the scan by calling the ScanFiles method

ScanFiles(path, SearchOption.AllDirectories);

});

}

private async Task PerformQuickScan(string path)

{

await Task.Run(() =>

{

// Perform the scan by calling the ScanFiles method

ScanFiles(path, SearchOption.TopDirectoryOnly);

});

}

private void ScanFiles(string path, SearchOption searchOption)

{

try

{

// Get the list of files in the specified path and search option

string[] files = Directory.GetFiles(path, "\*", searchOption);

int totalFiles = files.Length;

int scannedFiles = 0;

int detectedFiles = 0;

StringBuilder detectedFilesInfo = new StringBuilder();

// Iterate through each file and perform signature-based detection

foreach (string file in files)

{

UpdateProgressBar(scannedFiles, totalFiles);

string filePath = file;

if (signatureBasedDetection.IsPotentiallyMalicious(filePath))

{

// File is potentially malicious

detectedFiles++;

detectedFilesInfo.AppendLine($"file contain malicious hash : {Path.GetFileName(filePath)}");

detectedFilesInfo.AppendLine($"Path: {filePath}");

detectedFilesInfo.AppendLine();

}

scannedFiles++;

}

UpdateProgressBar(scannedFiles, totalFiles);

// Display the scan results

if (detectedFiles > 0)

{

MessageBox.Show($"{detectedFiles} file contain malicious hash .\n\n{detectedFilesInfo.ToString()}");

}

else

{

MessageBox.Show("No file contain malicious hash .");

}

DisplayScanSummary(scannedFiles, detectedFiles);

}

catch (Exception ex)

{

MessageBox.Show($"An error occurred during the scan: {ex.Message}");

}

}

private void UpdateProgressBar(int scannedFiles, int totalFiles)

{

// Calculate the progress percentage and update the progress bar

int progressPercentage = (scannedFiles \* 100) / totalFiles;

this.Invoke((MethodInvoker)delegate

{

progressBarscan.Value = progressPercentage;

});

}

private void DisplayScanSummary(int scannedFiles, int detectedFiles)

{

// Display the scan summary

string scanSummary = $"Scan complete!\n\nTotal Files Scanned: {scannedFiles}\nFile contain malicious hash: {detectedFiles}";

MessageBox.Show(scanSummary);

}

private void BtnSelect\_Click(object sender, EventArgs e)

{

// Open a folder browser dialog to select the folder for scanning

FolderBrowserDialog folderBrowserDialog = new FolderBrowserDialog();

if (folderBrowserDialog.ShowDialog() == DialogResult.OK)

{

textBox1.Text = folderBrowserDialog.SelectedPath;

}

}

private void Uc\_signaturue\_based\_Load(object sender, EventArgs e)

{

}

}

public class SignatureBasedDetection

{

private string signaturesFilePath;

public SignatureBasedDetection(string signaturesFilePath)

{

this.signaturesFilePath = signaturesFilePath;

}

public bool IsPotentiallyMalicious(string filePath)

{

try

{

string fileHash = CalculateFileHash(filePath);

string[] signatures = File.ReadAllLines(signaturesFilePath);

foreach (string signature in signatures)

{

if (signature.Equals(fileHash))

{

return true;

}

}

return false;

}

catch (Exception ex)

{

// Handle any exceptions during the detection process

Console.WriteLine($"An error occurred during the signature-based detection: {ex.Message}");

return false;

}

}

private string CalculateFileHash(string filePath)

{

using (var sha256 = SHA256.Create())

{

using (var stream = File.OpenRead(filePath))

{

// Compute the file hash using SHA-256

byte[] hashBytes = sha256.ComputeHash(stream);

return BitConverter.ToString(hashBytes).Replace("-", "").ToLowerInvariant();

}

}

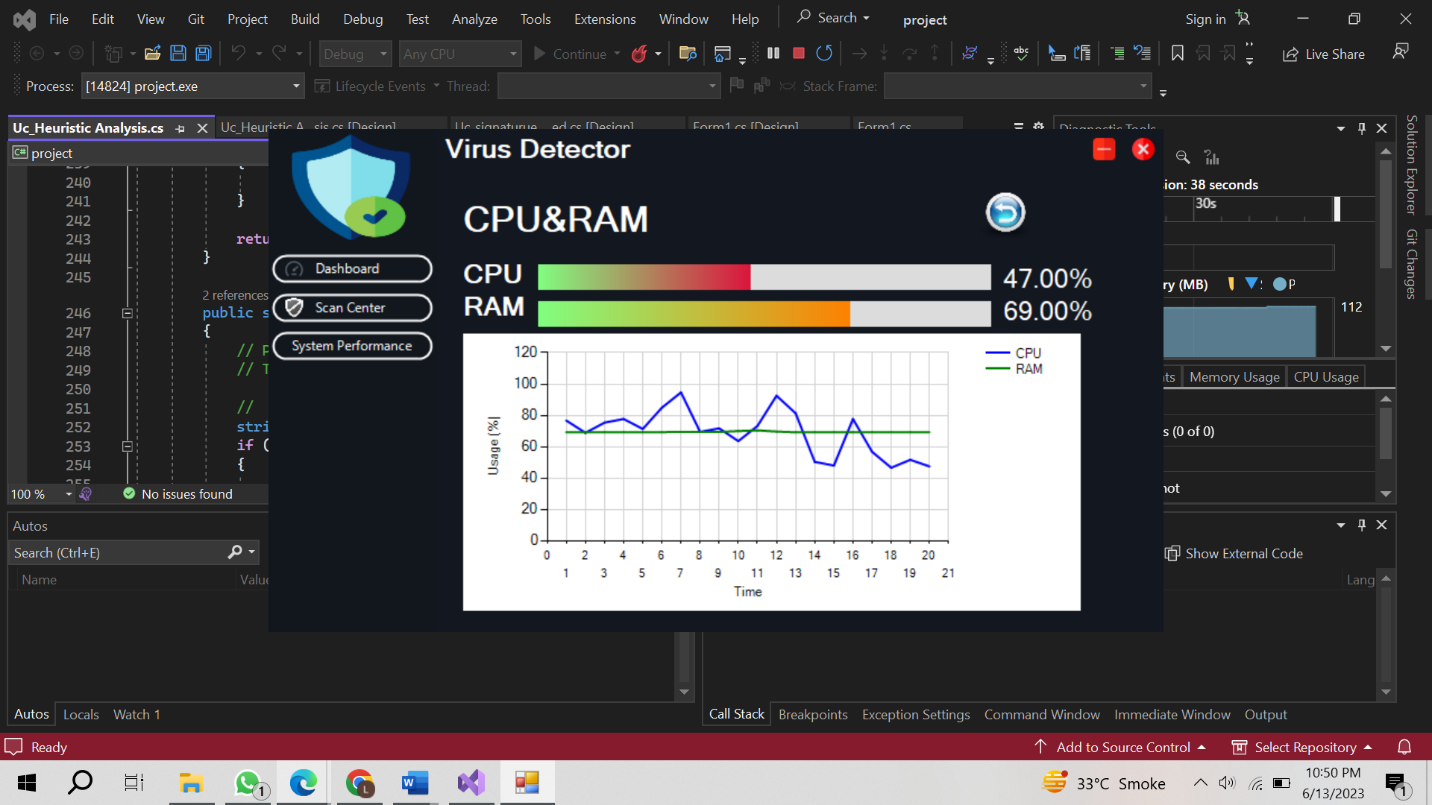
}

}

}

## System Performance:

I have developed a software application that provides users with real-time insights into the performance of their computer's CPU and RAM. With this software, users can monitor and analyze the crucial components that directly impact their system's overall performance.



## Code:

using project.Dashboard;

using System;

using System.Diagnostics;

using System.Windows.Forms;

using System.Windows.Forms.DataVisualization.Charting;

namespace project.performance

{

public partial class Uc\_Perform : UserControl

{

private PerformanceCounter cpuCounter;

private PerformanceCounter ramCounter;

public Uc\_Perform()

{

InitializeComponent();

InitializeCounters();

InitializeChart();

}

private void InitializeCounters()

{

cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "\_Total");

ramCounter = new PerformanceCounter("Memory", "% Committed Bytes In Use");

}

private void InitializeChart()

{

chart1.Series.Clear();

Series cpuSeries = new Series("CPU");

cpuSeries.ChartType = SeriesChartType.Line;

cpuSeries.BorderWidth = 2;

cpuSeries.Color = System.Drawing.Color.Blue;

Series ramSeries = new Series("RAM");

ramSeries.ChartType = SeriesChartType.Line;

ramSeries.BorderWidth = 2;

ramSeries.Color = System.Drawing.Color.Green;

chart1.Series.Add(cpuSeries);

chart1.Series.Add(ramSeries);

chart1.ChartAreas[0].AxisX.Interval = 1;

chart1.ChartAreas[0].AxisX.MajorGrid.LineColor = System.Drawing.Color.LightGray;

chart1.ChartAreas[0].AxisY.MajorGrid.LineColor = System.Drawing.Color.LightGray;

chart1.ChartAreas[0].AxisX.Title = "Time";

chart1.ChartAreas[0].AxisY.Title = "Usage (%)";

}

private void Uc\_Perform\_Load(object sender, EventArgs e)

{

timer.Start();

}

private void timer\_Tick(object sender, EventArgs e)

{

float cpuUsage = cpuCounter.NextValue();

float ramUsage = GetMemoryUsage();

UpdateCPUUsage(cpuUsage);

UpdateRAMUsage(ramUsage);

AddDataToChart(cpuUsage, ramUsage);

}

private float GetMemoryUsage()

{

float ramUsage = ramCounter.NextValue();

return ramUsage;

}

private void UpdateCPUUsage(float value)

{

int cpuUsage = (int)value;

gunaProgressBarCPU.Value = cpuUsage;

lbCPU.Text = $"{cpuUsage:F2}%";

}

private void UpdateRAMUsage(float value)

{

int ramUsage = (int)value;

gunaProgressBarRAM.Value = ramUsage;

lbRam.Text = $"{ramUsage:F2}%";

}

private void AddDataToChart(float cpuUsage, float ramUsage)

{

chart1.Series["CPU"].Points.AddY(cpuUsage);

chart1.Series["RAM"].Points.AddY(ramUsage);

if (chart1.Series["CPU"].Points.Count > 20)

{

chart1.Series["CPU"].Points.RemoveAt(0);

chart1.Series["RAM"].Points.RemoveAt(0);

}

}

private void label2\_Click(object sender, EventArgs e)

{

// Handle the label2 click event here if needed

}

private void gunaProgressBarCPU\_Click(object sender, EventArgs e)

{

// Handle the CPU progress bar click event here if needed

}

private void gunaProgressBarRAM\_Click(object sender, EventArgs e)

{

// Handle the RAM progress bar click event here if needed

}

private void lbCPU\_Click(object sender, EventArgs e)

{

// Handle the CPU label click event here if needed

}

private void lbRam\_Click(object sender, EventArgs e)

{

// Handle the RAM label click event here if needed

}

private void chart1\_Click(object sender, EventArgs e)

{

// Handle the chart click event here if needed

}

private void BtnBack\_Click(object sender, EventArgs e)

{

Uc\_dash uc\_Dash = new Uc\_dash();

uc\_Dash.Show();

this.Hide();

}

}

}