Project Proposal  
**1. Problem Description**  
To this day, basic memory corruption issues like buffer overflows, use after free, and stack overflows are a huge security nuisance for software systems. However, low level assembly code is especially tricky to identify and hot fix these vulnerabilities. Such cases are often handled poorly using conventional high level analyzers. The majority of existing tools are built on top of higher level languages or binary code analysis, and none provide effective memory corruption solutions for x86 and x64 architectures. Further, these tools often run into compatibility issues when being used on Linux based systems. A reliable and accurate static analysis tool necessary for enhancing software security. Such a tool is not just for finding vulnerabilities, but rather for finding them and then giving clear, actionable recommendations on how to fix them. Further enhancing the tool's ability to address known vulnerabilities is by integrating a comprehensive database, like the one we use, the CVE (Common Vulnerabilities and Exposures) database.

**2. Methodology**  
With the formalized structure of BugHex project, tool creation, dataset usage, validation, and the fine tuning of performance will all be made integral. First, the project will primarily involve creating a tool that is used to relate security weaknesses including buffer overflows, stack overflows, and use after free mistakes in x86 and x64 binaries type files on Linux based systems. That is why, more sophisticated methods such as control flow analysis and taint tracking will improve the tool’s ability to identify vulnerabilities in assembly code. Working with the CVE database will enhance the reliability of detected threats in connection with real-life cases and recommendations for their elimination. For the tool’s development into an even more efficient and accurate tool, the MalMem2022 dataset from the University of New Brunswick’s CIC will be used to recognize patterns and detect memory vulnerabilities, and while more datasets and CVE records will widen the range of the penetration of the tool.

The goals of the project are to produce a tool that loves at a rate of 1 second per one MB of assembly code while having better than 90% detection rate with as few false positives and negatives as possible. Other examples will be the preparation of elaborate reports that we shall use to present the discovered weaknesses and where they are within the component, together with recommendations of the most suitable measures for risk prevention based on data from CVE. Intended especially for the Linux systems, the tool will be tested with the real-life datasets such as MalMem2022 or CVE entries. As a result of the uses, the tool will focus on low level security, engaging with Penetration testers, Cyber security researchers and developers and will make practical recommendations on how certain risks have to be handled.

**3. Expected Results**  
The deliverables of the BugHex Project are supposed to develop a practical and operational static analysis tool that can insight memory corruption risks in the context of x86 and x64 binaries operating on Linux platform. It will be possible to detect such weaknesses as buffer overflows, stack overflows and, most importantly, use-after-free overflows, and provide recommendations on how to deal with these problems. Depending on the source of the link, it will offer further information such as descriptions of the existing threats, recommendations or ideas on how to respond to actual threats gathered in the CVE database. The tool will be made easily installable and usable for Linux systems to support various users with no restrictions on their abilities.

The rigorous testing will give the tool a minimum detection rate of over 90% thus providing a very low false negative and false positive rate. Future testing on datasets, including MalMem2022, will show how the tool copes with real security issues and is confirmed by CVE records. In conclusion, the attendees of the BugHex will include Penetration testers, cybersecurity researchers and developers of software, where it will assist in solving low-level security bugs and boost the safety of the applications.

 This research aims at filling this gap by developing dependable, high accuracy tool that has the capability of using CVE database, sophisticated techniques and effective datasets like MalMem2022 in identifying and preventing memory corruption vulnerabilities.

# **Project Name**

BugHex

# **Project Scope**

The goal of this project is to create a complete tool for penetration testers to discover memory corruption vulnerabilities in varied software applications' assembly code. The program will examine assembly code for the x86 and x64 architectures to detect typical memory-related issues such as buffer overflows, use-after-free mistakes, and stack overflows. Using static analysis techniques, the tool will deliver actionable insights for reducing security risks.

The scope involves employing actual cases along with example vulnerabilities to evaluate the tool's effectiveness, in addition to assuring seamless integration with numerous platforms such as Windows and Linux. While the research will focus on low-level assembly code vulnerabilities, it will not include high-level language vulnerabilities or non-memory-related exploits. This tool is intended to help penetration testers detect and repair major weaknesses in software.

# **Problem Statement**

Pent tester and researchers have lots of problems when it comes to memory corruption assessment and isolation, including methods for buffer overflows, use-after-free, and stack overflows. Modern approaches do not allow inspecting and analyzing low-level assembly code most of the time crucial to identifying these security threats. To this end, the project intends to create a new tool for performing static analysis of x86 and x64 files assembly code to improve software security.

As a result of using real-world cases and known vulnerabilities, the tool will be able to offer practical assessment and concrete recommendations. It must be correct, intuitive to operate, unobtrusive, yet essential during the penetration tester’s work, and exclude the high-level vulnerabilities.

**The seriousness of the problem:**

Why it is

important? to fix such memory corruption problems, such as buffer overflows and stack overflows remains an essential component of software security. These are problems through which hackers gain access to systems and other services making a huge security threat. Thus, knowing these issues, your tool might save you from an attack, it is very significant for maintaining safe and reliable computer systems.

Your tool deals with analysis of the low level assembly code in x86 and x64 files, level of detail that seems to be missed by most other security tools. This makes it absolutely priceless because it is able to LIIK – something that the other tools cannot do. Therefore, the identified gaps are resolved, and your tool is more comprehensive and effective, so that it can be considered a breakthrough in the sphere of cybersecurity and highly valuable for industries, which rely on safe software.

# **Objectives**

-The BugHex project is to develop an effective, comprehensive instrument intended for penetration testers, software developers, and cybersecurity researchers. Its responsibility is to identify and analyze the memory corruption vulnerabilities in assembly code of x86 and x64 and it is designed only for Linux OS and x86 and x64 architecture binary files. To this end, the tool will use sophisticated static analysis and analysis intents to focus on major threats, for example, buffer overflows, use-after-free type of errors, and stack overflows to improve software security.

-One of the major goals is, therefore, to make sure that the tool is compatible with Linux environments and offer reliable and effective services in vulnerability identification. The project also anticipates the relevance of real-world testing with real-practical case scenarios and existing exploits. This approach of validation reduces the gap between the pioneering of theoretical approaches to security tools, and actual proof that they can work as designed to eliminate security threats.

-Last but not least, it provides the specified goal with focus on user accessibility so that the tool will have an intuitive design that can help the user easily recognize and solve the issues. Here, making the tool more usable to cybersecurity professionals can be useful in the long standing goal of minimizing security threats in software applications.

-Whereas the success of the BugHex project will be determined depending on several factors The following are used to gauge the effectiveness of the BugHex project. The use of appropriately identified vulnerabilities will evaluate the detection accuracy and efficiency of the specified tool while checks on Linux systems will assess platform compatibility. The effectiveness of the proposed method in the real world will be proved and showed to the readers and the users’ feedback will compare the accessibility and usability of the method. Such measures help achieve the goal of the project that is devising an effective solution for detecting memory corruption vulnerability.

**Thematic Literature Review**

Advancements in Vulnerability Detection and Memory Corruption Analysis

Introduction

The increasing complexity of software systems, as well as the dependency on low-level programming languages, have increased the risk brought about by memory corruption vulnerabilities. Detecting and mitigating these holes is crucial to ensuring software security and dependability. This paper summarizes recent advances in vulnerability detection techniques, with an emphasis on static and dynamic analysis, deep learning approaches, as well as debugging tools, and emphasizes their importance to assembly-level analysis. The examined works are arranged according to time, from most current to oldest, to provide a historical overview of progress in this topic.

Recent Research on Vulnerability Detection

**Concurrency Vulnerabilities and Memory Corruption (2023)**

This paper addresses the issues provided by concurrency memory corruption vulnerabilities in both C++ and C systems, as well as detection and mitigation solutions. The researchers concentrated on discovering concurrency vulnerabilities, that involve data races and synchronization issues and presented the Conzzer fuzzing tool for dynamic analysis.

Conzzer successfully discovered concurrency concerns using context-sensitive coverage metrics and adjacency-directed modification methods. The findings revealed that static analysis gave rapid identification but lacked accuracy, whereas dynamic techniques such as Conzzer provided improved accuracy at a higher processing cost. The investigation revealed that tackling concurrency vulnerabilities in real-world circumstances requires a balanced strategy that includes both static and dynamic solutions.

**Public Enclave Security: Host-to-Enclave Boundary and Memory Corruption Vulnerabilities (2023)**

This study investigates vulnerabilities at the host-enclave boundary in Intel SGX public enclaves. To better understand typical flaws, the study examined API implementations, threat models, and data management techniques. It highlighted incorrect API implementations as a significant source of vulnerabilities, allowing attacks such as enhanced return-oriented programming (ROP) and non-control data exploitation. SGX's encryption and integrity checks proved effective, but insufficient against these sophisticated attackers. The analysis stressed the importance of enhanced API  implementation and extensive protection methods in order to increase enclave security and eliminate boundary vulnerabilities.

**Deep Learning for Binary Executables Without Source Code (2022)**

This work investigates the use of deep learning to find vulnerabilities in binary programs. The researchers encoded binary data into LLVM Intermediate Representation (LLVM IR) and trained recurrent neural networks including GRU, LSTM, and SRNN for binary and multi-class

classification. The findings demonstrated 88 percent accuracy in binary classification and 77 percent in multiclass classification, illustrating the difficulty in discriminating between comparable vulnerabilities such as buffer overflows and underflows. The study highlighted the potential of deep learning for binary analysis, while also emphasizing the need for more datasets and improved model architectures to increase classification accuracy.

**Software Vulnerability Detection Based on Binary Intermediate Segmentation (2021)**

This study presents a system for detecting software vulnerabilities that employs binary intermediate segmentation and deep learning techniques. Employing Word2Vec to segment binary code into vectors, the technology made it easier to train deep learning models to detect vulnerabilities. The results showed a reduction in detection time and potential increases in accuracy, but the study lacked comparisons directly to existing approaches and real-world confirmation. The research emphasized the framework's promise while emphasizing the importance of various datasets and extensive testing to establish practical applicability.

**A Hybrid Vulnerability Detection System Using Assembly and Source Code Analysis (2021)**

This paper presents a hybrid detection approach that uses both assembly and source code analysis to improve vulnerability detection accuracy. To examine hybrid code slices, the researchers used

a deep learning model based on hyper fusion that combines early and late fusion methodologies. The hybrid model reached up to 97% accuracy, surpassing single-mode detection approaches and identifying vulnerabilities within either assembly or source code. However, the study has limitations resulting from its reliance on pre-labeled datasets and the difficulty in extending the

system for cross-platform contexts. The study showed the advantages of mixing multiple modalities while underlining the significance of cross-platform capabilities.

**MemSherlock: Automated Debugging for Memory Corruption (2020)**

This paper introduces MemSherlock, a technique for detecting memory corruption issues such as buffer overflows. MemSherlock gave a comprehensive perspective of memory corruption issues by combining static and dynamic analysis, as well as taint tracking. The tool underwent testing on 11 real-world applications and showed good accuracy in detecting corruption locations and tracing input propagation channels. The investigation acknowledged problems in dealing with embedded

assembly code and inadequate library specifications but stressed the importance of combining static and dynamic techniques for effective vulnerability detection.

**Discovering Buffer Overflow Vulnerabilities in the Wild (2018)**

This research investigation looks at the identification, reporting, and utilization of buffer overflow vulnerability. The researchers examined data from 58 editors, concentrating on the tools and methodologies used to identify vulnerabilities. Fuzzing and debugging utilities, including SPIKE and Immunity Debugger, were found to be effective. Reporting techniques vary, with some advocating full disclosure and others preferring coordinated measures to facilitate quick patching. The investigation identified shortcomings in the use of static analysis tools and emphasized the importance of systematic detection approaches for improving vulnerability management.

**A Very Effective Debugging (VED) Technique for Runtime Error Recovery in .NET Applications (2017)**

The present study describes the VED technique for identifying and recovering through runtime mistakes, particularly division by zero issues, in.NET applications. VED used Immunity Debugger and Python scripting to automate error detection and recovery operations, decreasing debugging complexity and increasing program reliability. The approach offered user-friendly recovery choices via a graphical interface. However, its usefulness was restricted to division by zero errors, and its dependency on out-of-date technologies such as Python 2.7 caused compatibility issues. The analysis recognized the utility of lightweight debugging tools while emphasizing the requirement for broader applicability and updating.

Reported gaps and contributions to this research

Several important deficiencies were identified in the examined studies. Particularly, the research is primarily focused on high-level or binary code analysis, with little emphasis on vulnerabilities particular to assembly code. Static analysis, although its importance, is neglected and frequently loses precision for low-level vulnerabilities. Furthermore, several of the frameworks and tools evaluated lack cross-platform capabilities and fail to prove their ideas in real-world settings, instead depending on controlled datasets. Advanced memory corruption issues, which include

stack overflows and use-after-free faults, are also not well addressed in current tools and approaches.

How This Research Addresses the Gaps

Addressing Gaps with This Research This research focuses on an important but often disregarded area: assembly-level vulnerabilities in binary files, notably for x86 and x64 architectures running Linux systems.  It utilizes static analysis methods which are aimed solely at identification of the most urgent threats like buffer overflows, stack overflows, use-after-free, etc. In contrast with many existing tools, which might not deliver detailed analysis at the assembly level, this tool improves the program’s ability to function in Linux environments. Additionally, the project continues the tradition of focusing not on the construction of models but on their application. Employing these explicit appraisals benefits from translating theoretical ideas into practice because it optimizes the tool’s accuracy and real-world effectiveness in detecting memory corruption related flaws.

System components, functionality and Interface (Wireframes / Mockups / Prototype)

**Define Major Components:**

The system consists of five major components: The Input Module, Disassembler, Static Analyzer, CVE Validator and Report Generator. They are coupled together to detect and analise assembly code based vulnerabilities of memory corruption. Together, they make up part of the core architecture of the tool: to allow penetration testers to quickly find their way and identify potential threats.

**Describe Each Component:**

* Input Module:

The input module is the entrance of the system. They provide a user the upload of either binary file or assembly file. The module directs uploaded file to Disassembler if the file is in binary format. It's because of this component that the tool can handle various file types and therefore be universal as to different testing scenarios.

* Disassembler:

It’s this component that reads in binary files and converts them into human readable assembly code. Reverse engineering technique are used by the Disassembler to decompile binary formats to the low level instructions. It is crucial for the subsequent analysis performed by the Static Analyzer, since it is assembly code that Static Analyzer is focused on.

* Static Analyzer:

The system's heart is the Static Analyzer, which is to find vulnerabilities in assembly code. It uses static analysis technology that scans the code for ordinary memory corruption problems, such as buffer overflows, stack overflows, and use after free errors. Additionally, it shows the affected part of the code and therefore suggests a mitigation.

* CVE Validator:

As reported by CVE Validator, the CVE Validator checks detected vulnerabilities with existing CVE (Common Vulnerabilities and Exposures) database entries. On top of that, this module offers additional validation by linking up discovered vulnerabilities to relevant CVEs, bringing additional context and a prioritization hierarchy for fixing these issues. This step can be enabled or disabled depending on the users behavior.

* Report Generator:

The Static Analyzer and CVE Validator output results that are compiled into a report using the Report Generator. This report details identified vulnerabilities, their severity and potential mitigations. It ensures that penetration testers receive a well-structured and actionable output they can use to help out with their security efforts.

Block diagram:

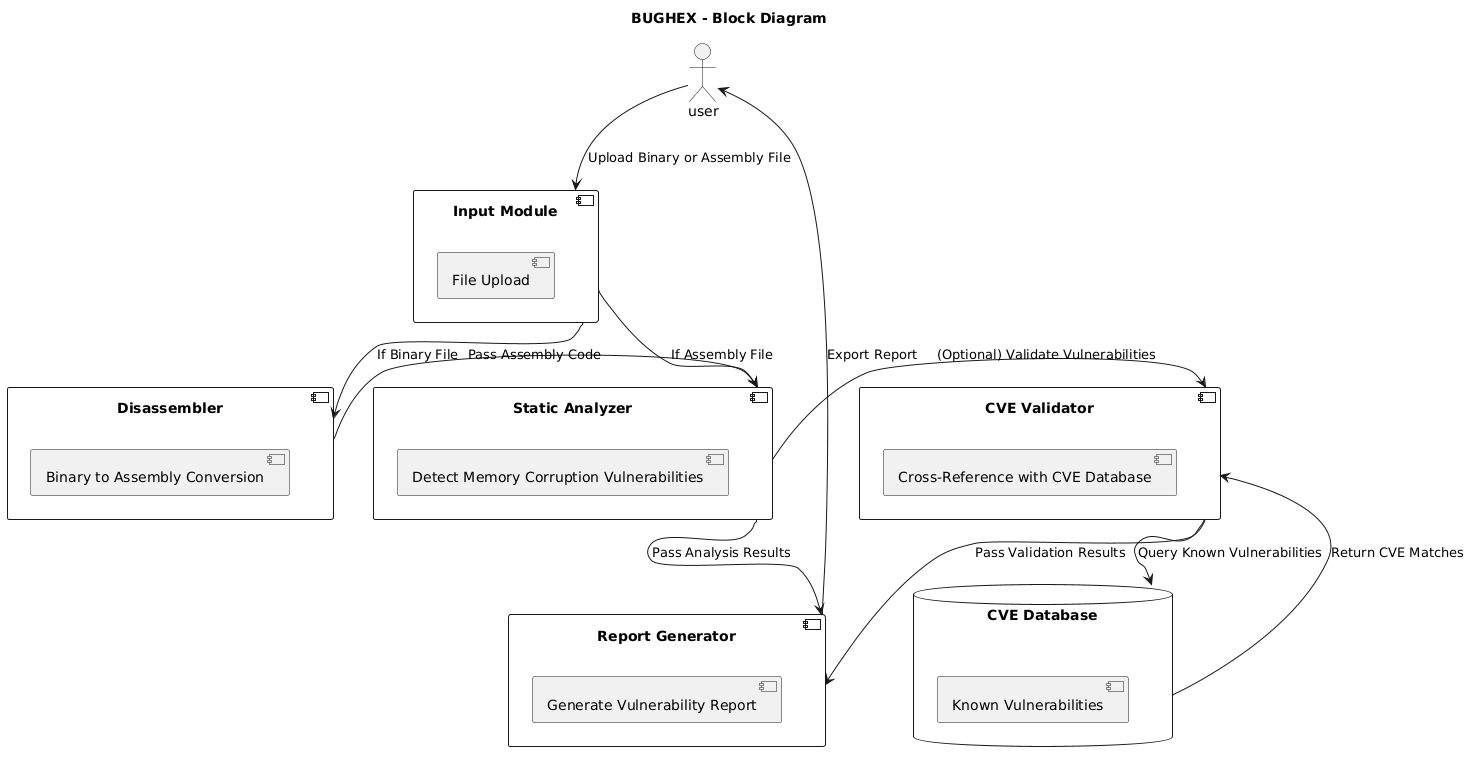


Figure (1) Bughex block diagram

List Key Functionalities:

The Memory Corruption Detection Tool is intended to perform several important (and relevant to its mission of detecting memory flaws in assembly code) operations. These key functionalities include:

* File Input Handling:

Another requirement must be that the system should enable users to upload binary or assembly files. It should check the file type and also if it is a binary file then transform them into readable assembly code using disassembler.

* Disassembly:

With binary files, the system has to decompile the binaries into assembly code so as to allow subsequent static analysis. This step is important because all types of executable files can be analyzed, as will be described below.

* Static Analysis for Vulnerability Detection:

The basic capability is to parse the ASM code to flag memory-related issues including but not limited to; buffer overflows, stack overflows and use after free. A perfect tool should be able to give out information about the vulnerabilities such as the type of vulnerability, the extent of the vice and where it is located in the code.

* CVE Validation (Optional):

It was necessary to make the use of this feature optional, but the tool must allow the user to check if detected vulnerabilities match some records from the Common Vulnerabilities and Exposures (CVE) database. Such validation is important to determine if the detected problems are already existing vulnerability.

* Report Generation:

Lastly, the system should be able to present an exhaustive report that gives the analysis’s results. It is advised that the report should highlight facts about the vulnerability, its level of risk associated, and steps that might be taken to address the issue of vulnerability and the report can include link to CVEs if any.

* User Interface and Interaction:

Penetration testers need a friendly UI for working with the tool, loading files, adjusting settings (such as turning the CVE check on/off), and reading or downloading the generated report at the end.

* Error Handling and Feedback:

The tool should be able to politely navigate between errors like, unsupported file formats or corrupted files, and must have some informative responses for the user.

Map Functionalities to Components:

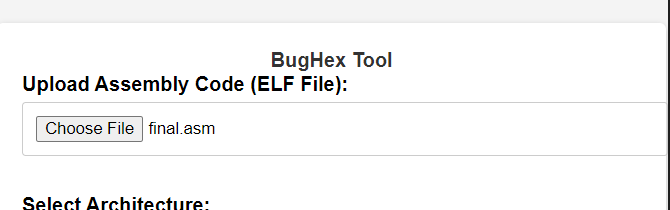


Figure (2) file input component

Input Module:

File uploads are handled by the Input Module, this is to enter binary or assembly files using the user. It does file type detection to know whether the uploaded file is binary or already in assembly format. It also returns feedback for unsupported file types or corrupted files and tells them if anything goes wrong with the file they supply.

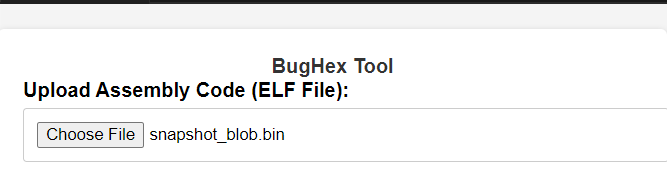


Figure (3) disassemble the binary to assemble file

Disassembler:

Disassembler converts binary files into assembly code which is later scrutinized. It both ensures compatibility with x86 and x64 architecture, making the tool able to process a wide variety of applications. After the disassembly is complete, it forwards the resulting assembly code to the Static Analyzer for vulnerability detection.



Figure (4) static analyzer result

Static Analyzer:

It is the job of the Static Analyzer to look through the provided assembly code to find memory related vulnerabilities such as buffer overflows, use after free errors and stack overflows. It shows how each discovered vulnerability is described, type, where it was discovered, and what severity it has. Identifying potential security risks in low level code is a crucial task: this component is a significant utility in taking care of this.

Figure (5) CVE ID and CVE reference

CVE Validator:

The CVE Validator method cross references detected vulnerabilities to the Common Vulnerabilities and Exposures (CVE) database. It also provides validation results that show whether known vulnerabilities were identified and existing records of the issue can be found. CVE validation is optional and should be enabled on environments having access to external databases.

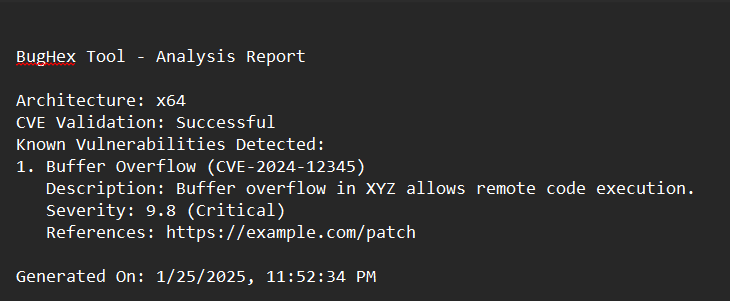
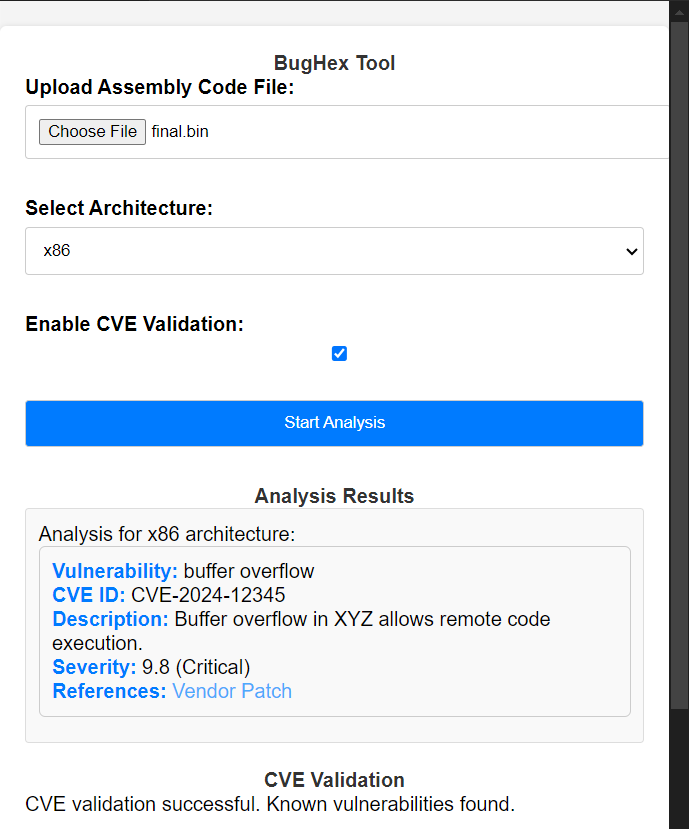
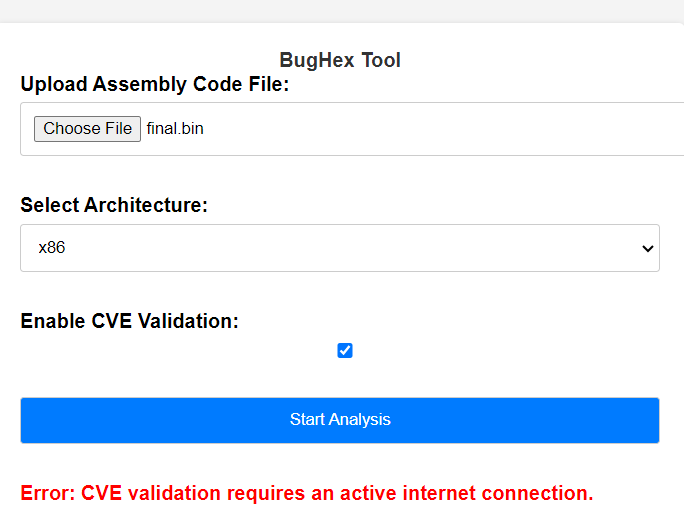


Figure (6) Bughex Analysis report

Report Generator:

All findings are combined into a report structured and comprehensive for the user by the Report Generator. It outlines all of the trivial details associated with detected vulnerabilities: type of vulnerability, location of vulnerability, severity of vulnerability and suggestions on how to mitigate a vulnerability. The report also refers to CVEs if CVE is applied. The report you generate can be export into different format like PDF, HTML or plain text.





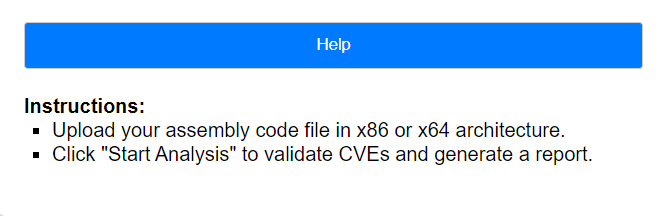


Figure (7) User Interface

User Interface (UI):

It gives Penetration Testers an intuitive way to interact with the tool. It allows file uploads, configuration options (i.e. enable or disable CVE validation), and downloaded generated reports. The UI also provides real time feedback and error messages to assist the use in the tool’s operation.

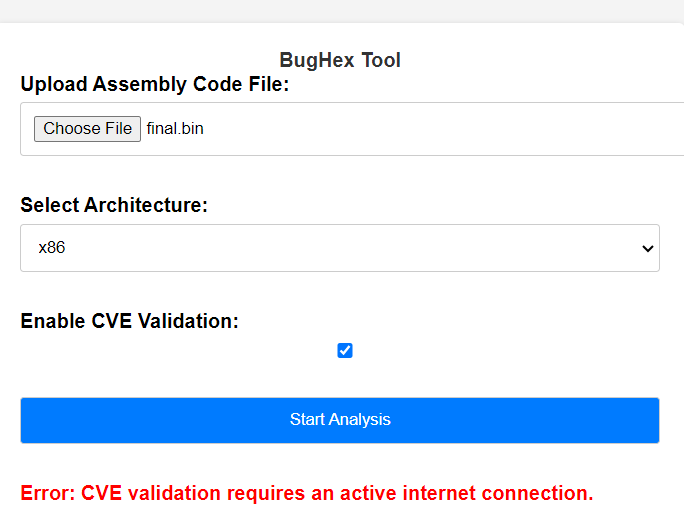
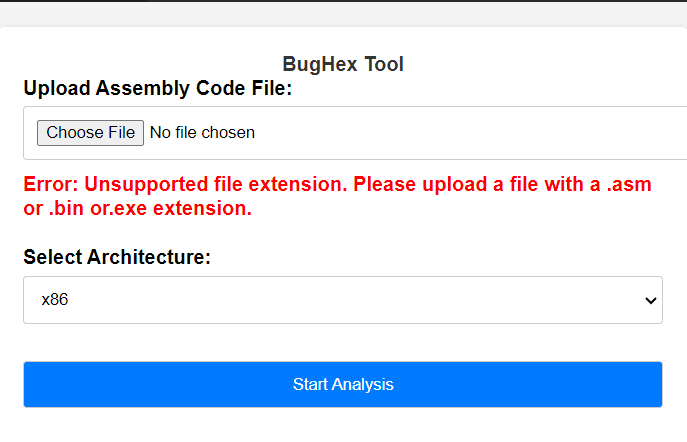


Figure (8) error handling

Error Handling Module:

The Error Handling Module handles errors throughout the system of what files support, such as database access, disassembly etc. The UI ensures that user friendly feedback is displayed when any problems occur with the tool operation through the UI as it helps the users understand and sort out the issue at hand.

Existing Similar Solutions (tools):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Tool | Platform | Primary Use | Key Features | Strengths | Weaknesses |
| GDB with PEDA | Linux, macOS | Debugging, Reverse Engineering | Python scripting, memory inspection, pattern searching, symbolic execution | Powerful, highly customizable, open-source | Steep learning curve, complex setup |
| Immunity Debugger with Mona.py | Windows | Exploit Development | Pattern generation, memory corruption detection, automated exploitation | User-friendly interface, powerful scripting, focused on Windows exploits | Proprietary, limited to Windows |
| Radare2 | Linux, macOS, Windows | Reverse Engineering, Vulnerability Analysis | Disassembly, debugging, scripting, analysis of binaries | Versatile, open-source, highly customizable | Steep learning curve, complex interface |
| Binary Ninja | Linux, macOS, Windows | Reverse Engineering, Vulnerability Analysis | Powerful analysis engine, user-friendly interface, scripting | User-friendly, powerful analysis capabilities, commercial support | Commercial license |
| Angr | Linux, macOS, Windows | Binary Analysis, Vulnerability Research | Symbolic execution, taint analysis, static and dynamic analysis | Powerful analysis engine, open-source, academic focus | Complex setup, requires programming skills |
| Cutter | Linux, macOS, Windows | Reverse Engineering, Vulnerability Analysis | User-friendly GUI for Radare2 | Intuitive interface, built on top of powerful Radare2 engine | Limited to Radare2's capabilities |
| MemSherlock | Memory Corruption Vulnerability Detection | Automated detection, malicious payload analysis | Automated detection, malicious payload analysis | Automation, Proactive detection, Deep analysis | False positives, Limited scope, Complexity |

Figure (9) similar tools Table

Techniques of these tools:

GDB with PEDA combines the functionality of the GNU Debugger (GDB) and Python Exploit Development Assistance for GDB (PEDA). GDB serves as a powerful debugger API that communicates with the target process to start or stop operations, read or write memory, and set breakpoints. It also features a scripting language that can be further exploited to automate tasks and enable customized analysis. PEDA enhances GDB by leveraging Python’s robust scripting capabilities, enabling feature enhancements and operational automation. It utilizes GDB's API to interact with the debugger and retrieve information about the target process, such as memory maps, registers, and disassembled code.

Immunity Debugger, when paired with Mona.py, provides a dynamic debugging environment. The debugger's API allows users to observe and manipulate the interactions within a target process. Its scripting language supports extending functionality and automating specific tasks. Mona.py enhances the debugger by using Python scripting for operations like pattern generation, memory analysis, and exploit creation. It communicates with Immunity Debugger's API to control processes, analyze memory, and set breakpoints, offering a powerful toolkit for exploit development.

In fact the go to tool for reverse engineering is radare2 which is a complete framework that includes functionalities in disassembly and debugging. The reverse engineering core comprises of complicated formula that disassembles binaries into assembly instructions and debugs code line by line. It also provides script support for automation and customization using languages such as Python. Static and dynamic kinds of analysis are also available in the analysis engine, it will allows users to examine binaries without executing these binaries to look for weaknesses or allows users to run these binaries to monitor its behavior, and gather data on capabilities and memory.

Binary Ninja is another kind of reverse engineering which has its main strength in disassembly and analysis. It uses sophisticated mathematical algorithms for conversion of binary code into assembly commands and have integrated analysis for function, data and control flow recognition. It supports the graphical user interface that can be used uniformly for viewing patterns and decoding of the binary code. Just like most other software that are designed for reverse engineering, Binary Ninja also provides scripting for the purpose of automation as well as customization of the analysis.

Angr is a tool oriented to symbolic execution and static analysis. Its symbolic execution engine computes all feasible executions of a program by treating the input values as symbolic variables, whereas constraint solvers examine variable values in different points of the program’s computation. When it comes to static analysis Angr provides control flow analysis for finding program vulnerability and data flow analysis for finding data leaks or security issues.

Cutter which is based on the Radare2 framework comes with the trigger from Radare2 for disassembling, debugging, and analyzing. It differentiates itself from the rest by depicture a graphical front end that makes it easier to utilize the many features of Radare2.

# Target Audience

1. **Penetration Testers**: For identifying and detailing low level risks.
2. **Cybersecurity Researchers**: Especially for memory corruption investigation and for enhancing the strategies used in identifying such threats.
3. **Software Developers**: For visiting or reviewing their possible areas of security vulnerability in their software at various stages of the development process.

# Requirements (Functional and Non-Functional)

**Functional Requirements**

* **Assembly Code Analysis**: This tool has to be learning assembly code for the x86 and x64 architectures. It must identify common memory corruption vulnerabilities including; buffer overflows, use-after-free, and stack overflows.
* **Static Analysis**: The tool needs not run the assembly code, and, therefore, has to use static analysis to analyze the code. It should produce a report that outlines some of the vulnerabilities opened up on the web applications, the type of vulnerability and areas of code broken down by section.
* **Multi-assembly files types Support**: The tool must be able to easily work with the disassembling of any type of assembly code including x86 and x64
* **User Input and Configuration**: The tool has to enable the uploading of assembly code by the users. Users should be able to set analysis parameters regarding the kinds of vulnerabilities that should be searched for, or the architectures to consider.
* **Validation Using Known Vulnerabilities**: It should compare its detected results with the known vulnerabilities and should give some references to the number of CVEs.
* **Output Generation**: The tool must be able to offer recommendations for handling the risks which have been realized. Output reports should export to file formats in portable file formats such as a text file.

**Non-Functional Requirements**

* **Performance**: The tool must be capable of scanning code in a relatively short amount of time, ideally, it should scan average binaries such as, 1MB size of assembly code within say, 10sec or even less.
* **Accuracy**: Sophisticated static analysis tool must also have a very high accuracy about detecting memory corruption such as 90% and above. To be reliable it should allow for few false positive predictions and few misdiagnoses or false negative predictions.
* **Scalability**: The tool has to do large binaries and assembly code and has to okay not be slower.
* **Usability**: It should also be easy to use, pleasant for a variety of levels of technical proficiency in penetration testers. It should be clear what errors occurred and there ought to be accessible help contents.
* **Portability**: The tool should be operable under most used operating systems including Linux. It has to not be very complex or have many a priori conditions that need to be established.
* **Security**: The tool has to guarantee that an assembly code uploaded for processing will not pose a threat to the user’s system. It should not store any sensitive code and the analysis results in this case without the user consent.
* **Extensibility**: The tool should be modular so that future enhancements are easy to incorporate for instance the addition of other types of vulnerabilities and other architectures.
* **Documentation**: Much documentation needs to be written for installation instructions, manuals for users, and documentation for developers.

Constrains

Technical Constraints  
The use of BugHex has certain technical issues in the progress of its development. Developing static analysis algorithms in Python them deemed necessary is complex, particularly for constructing a highly extensible and efficient tool as well as utilizing natural language processing models to process assembly code. This will also take some time and energy of the team. Another of them is the identification of the CVE database, where it is important to know how this database is constructed and updated, as well as, probably, with which method it is better to work. Such technical activities may cause productivity decrease on this project while we struggle through these challenges.

Resource Constraints  
Our project is constraint by time and availability of team members that would complete the work on time. Coming up with analysis algorithms, linking the CVE database, and proving real life scenarios require much hard work, which may be hard if one is working singly. Another problem is the scarcity of real-life applications and cases of certain kinds of vulnerabilities and threats. This might necessitate consulting other departments such as cybersecurity specialists or using a little longer to source data to perform the tests. These limits may limit progress and the number of tests we can conduct to a significant extent.

[Gantt Chart:](file:///C:\Users\User\Downloads\BugHex%201.pod)

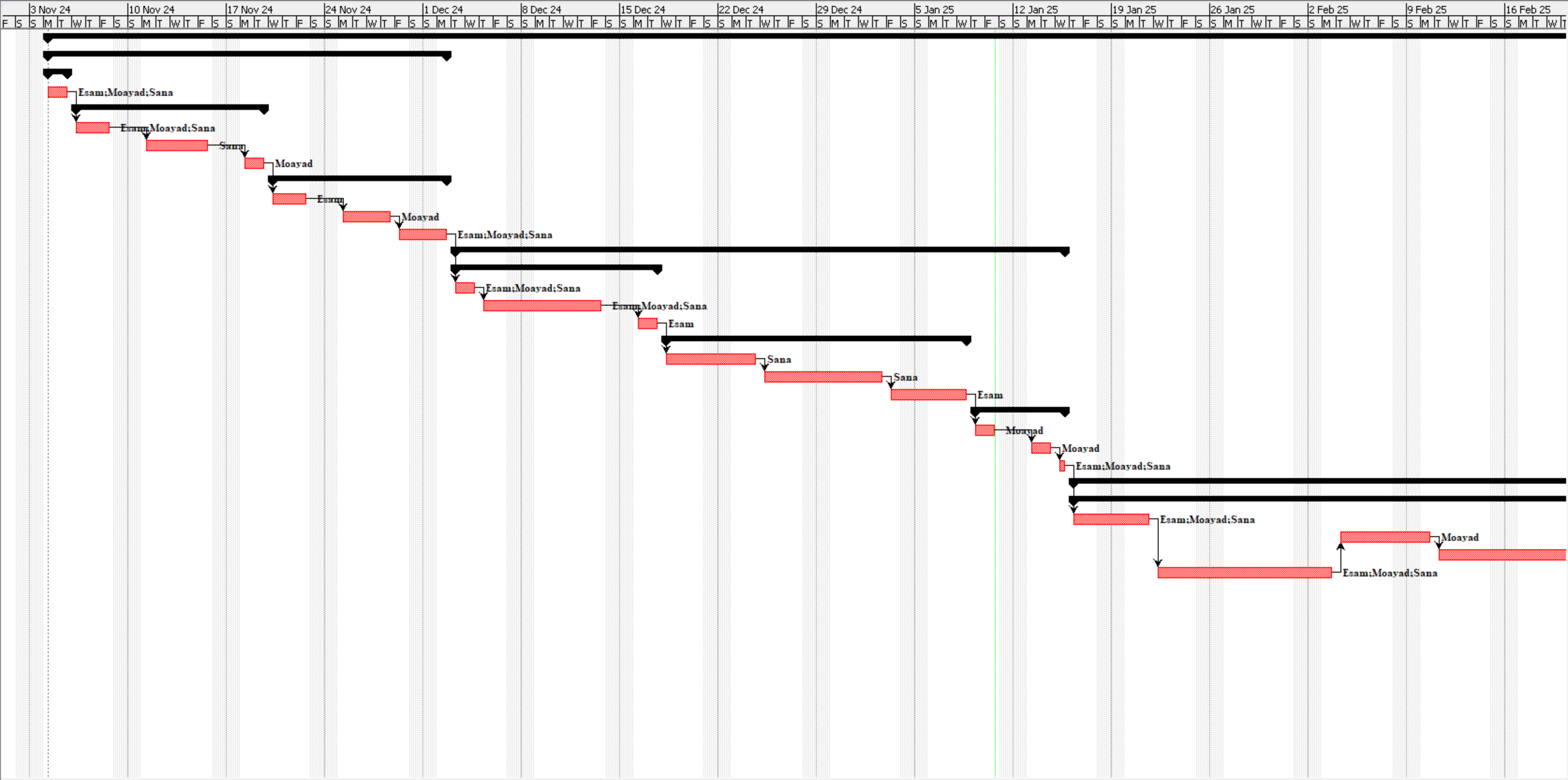


Figure (10) Gantt chart

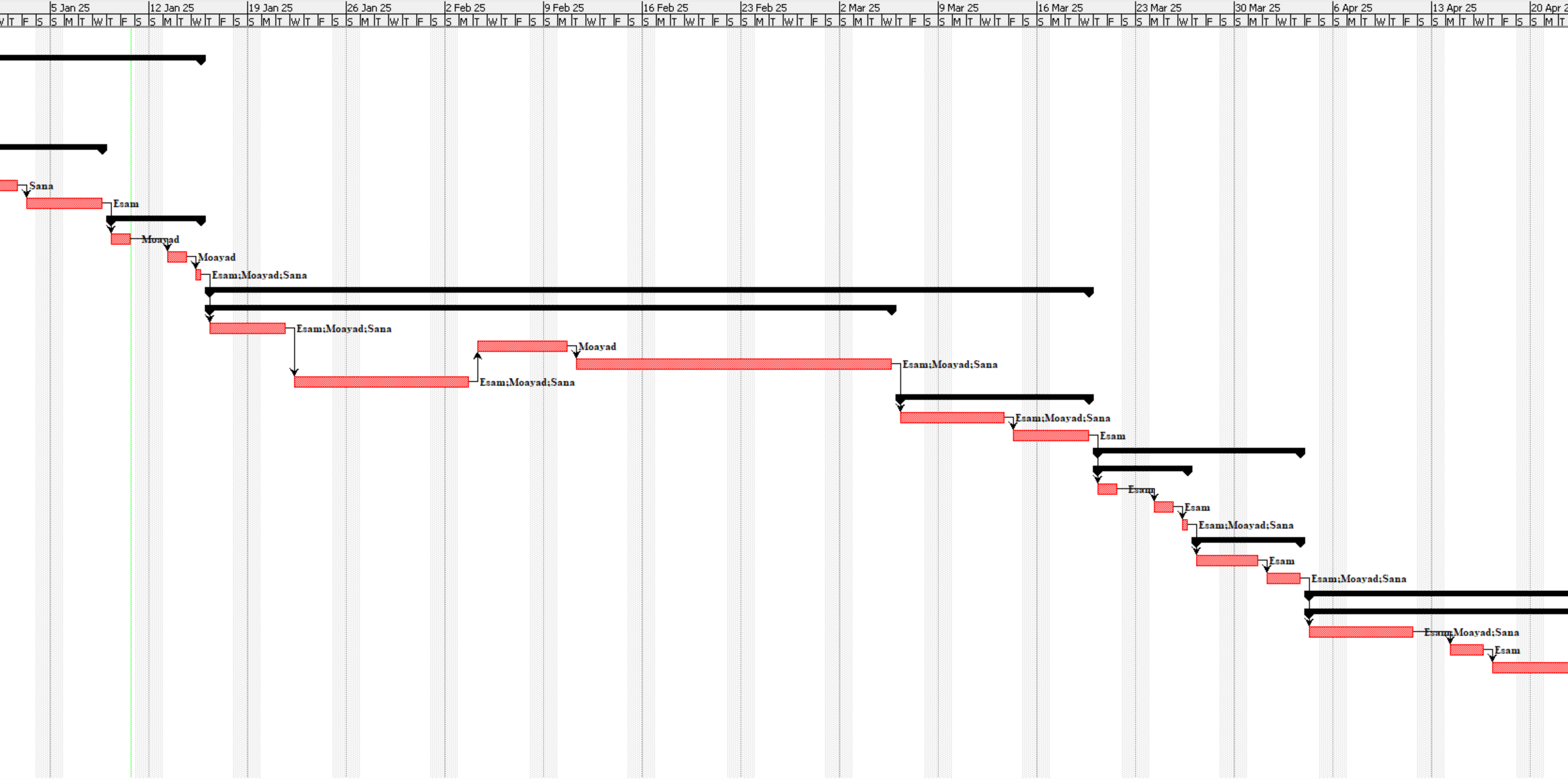
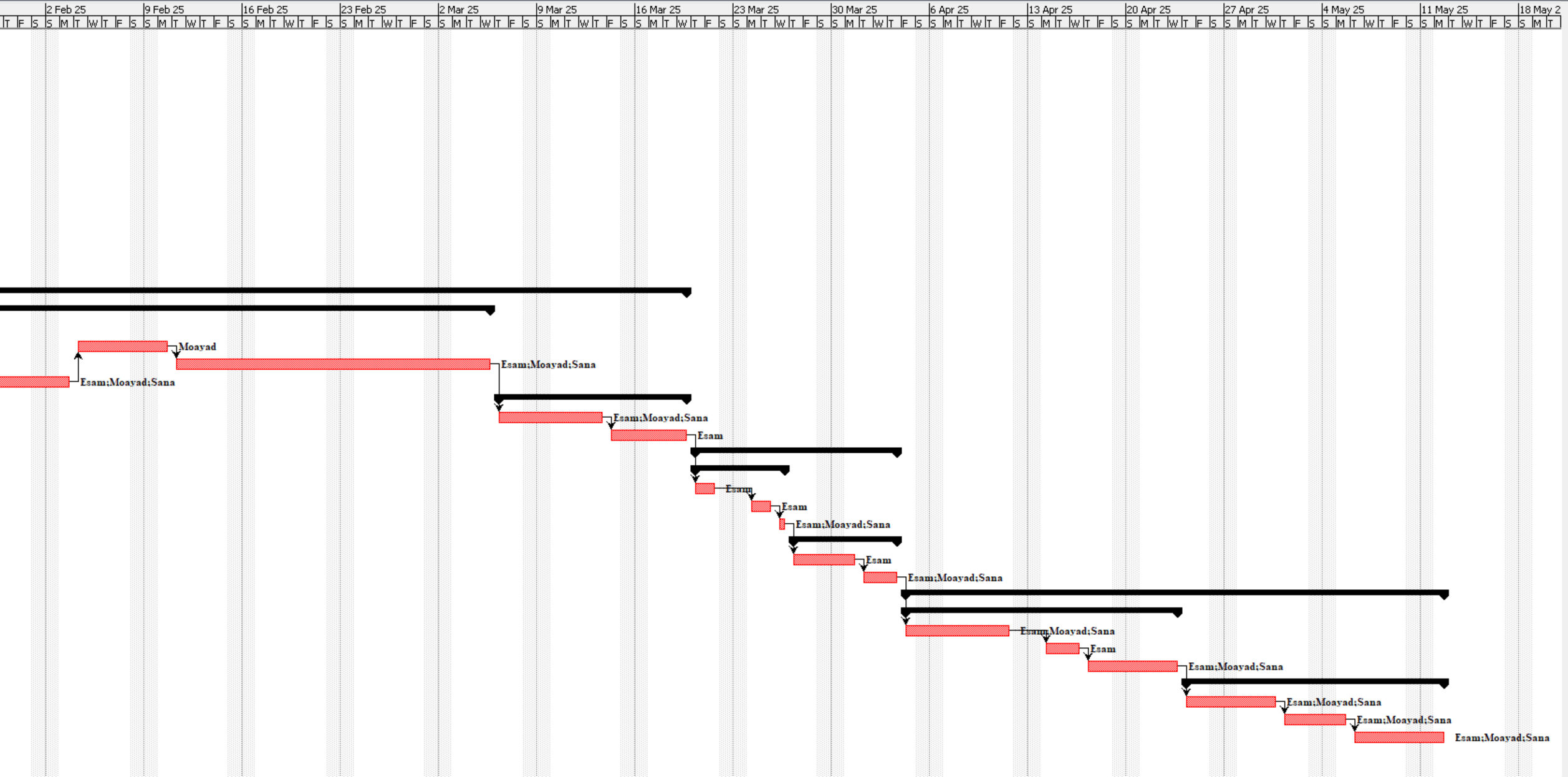


Figure (11) Gantt chart

 Figure (12) Gantt chart

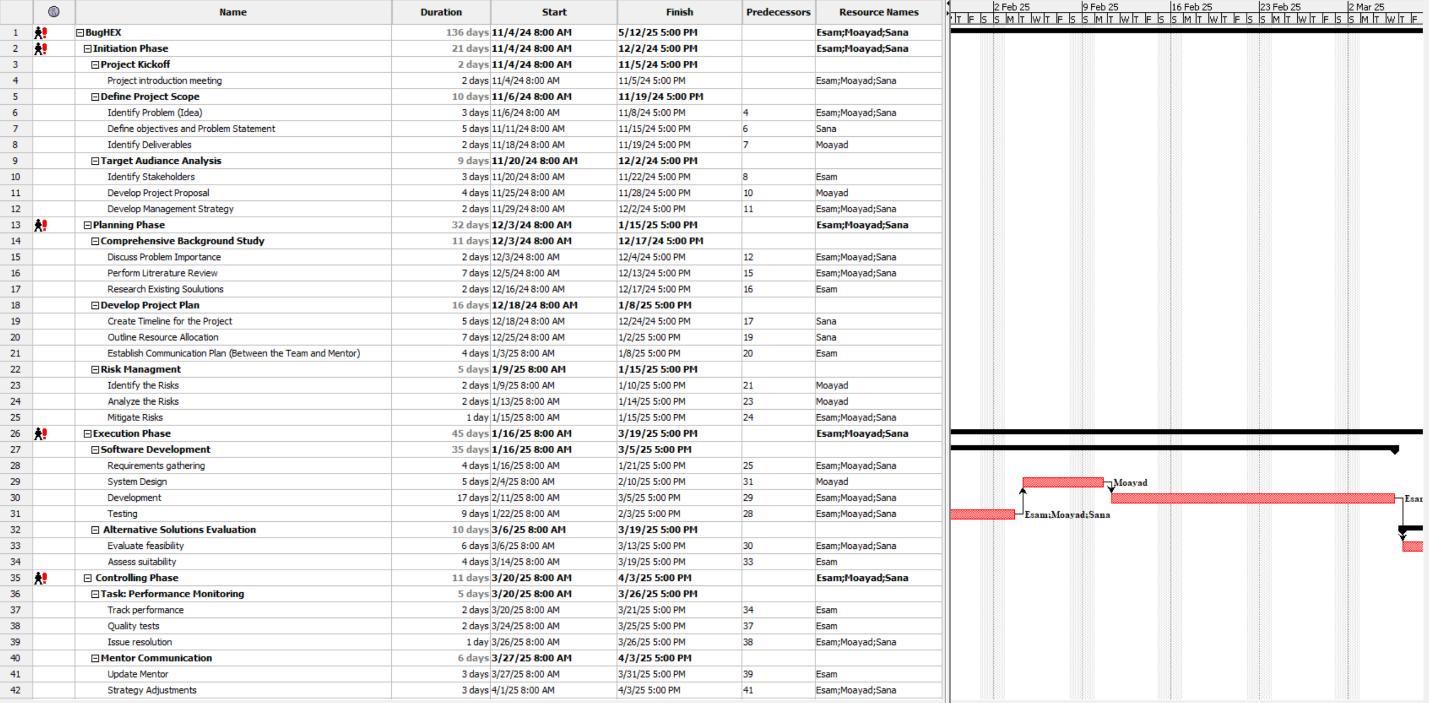


Figure (13) Gantt chart

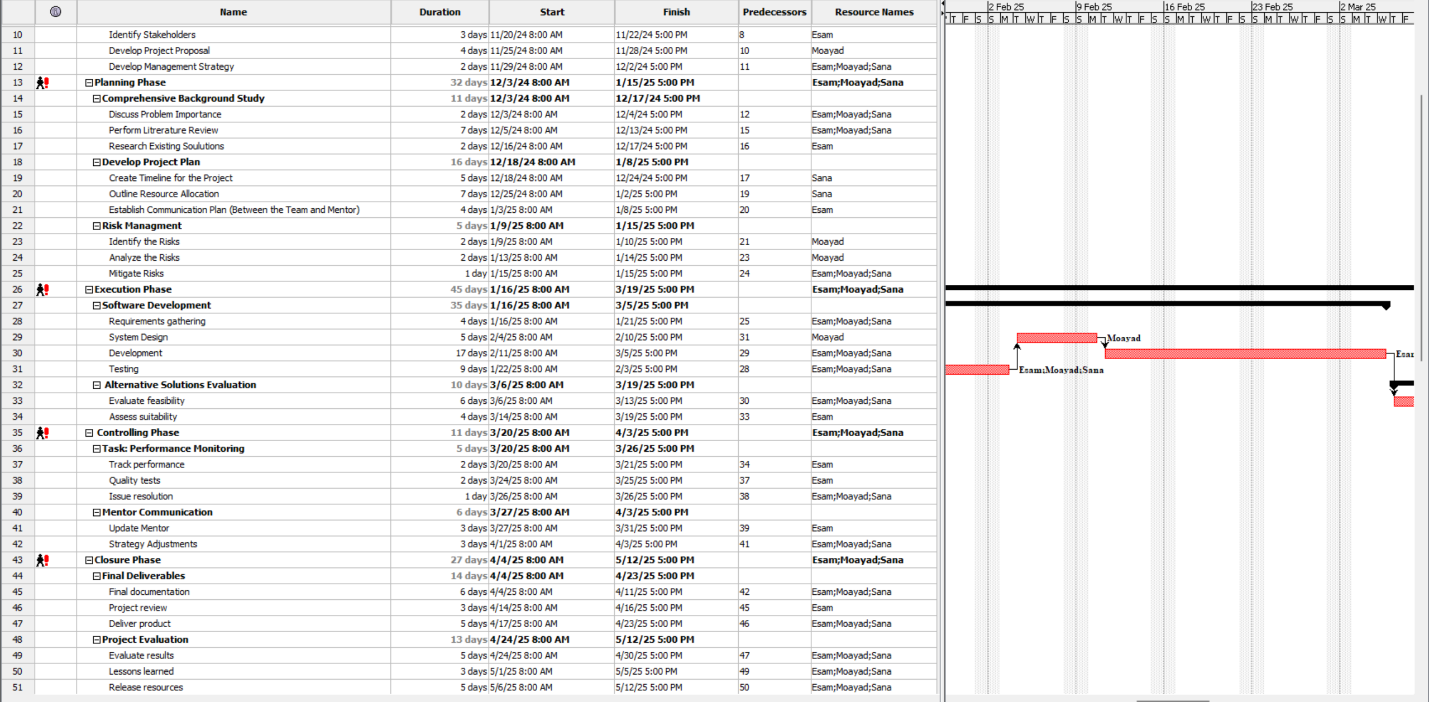
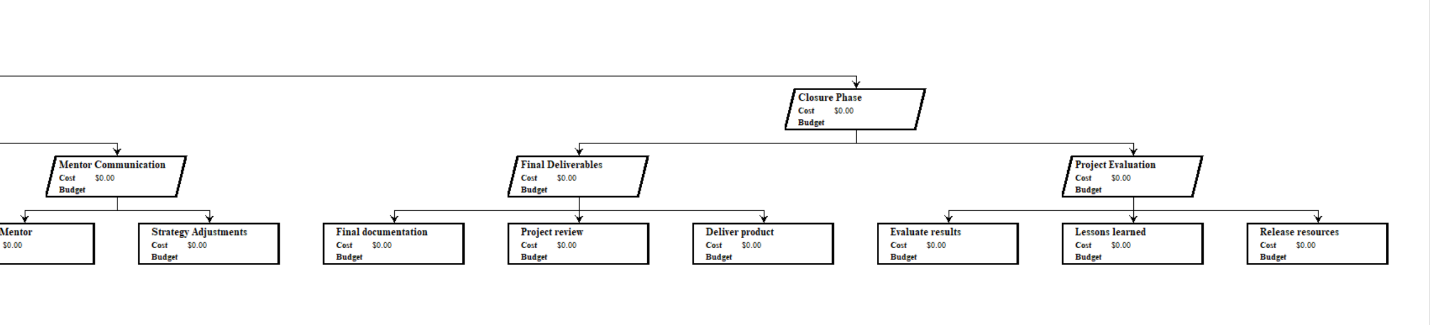
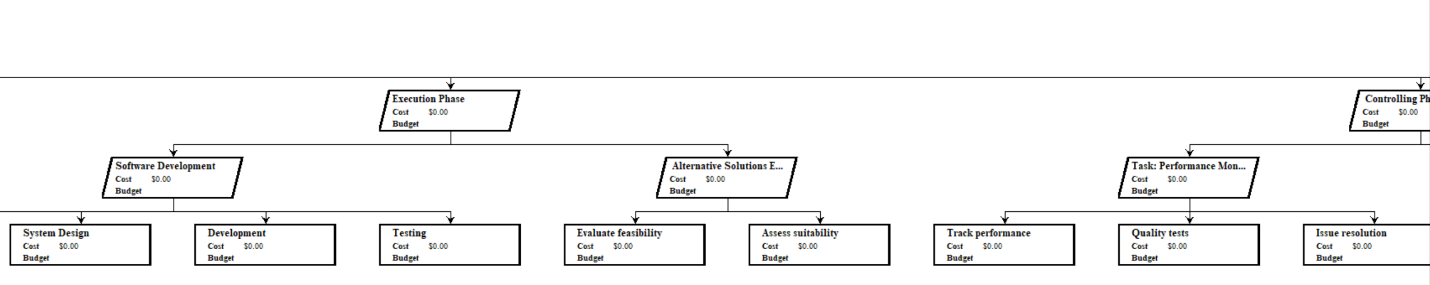
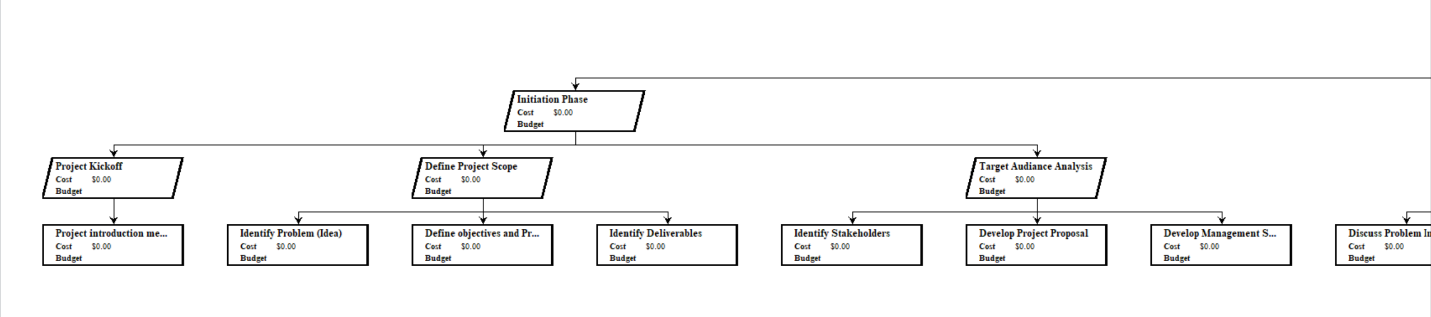


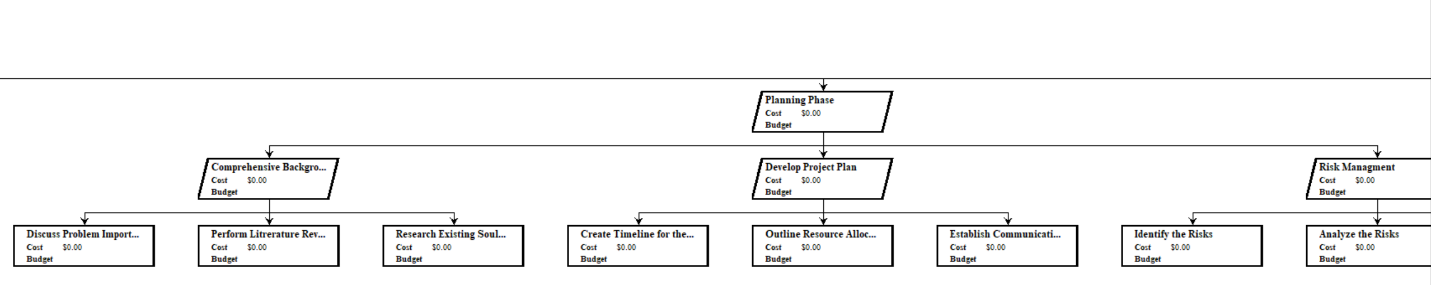
Figure (14) Gantt chart

WBS:



Figure(15)WBS 

Figure(16)WBS 

Figure(17)WBS 

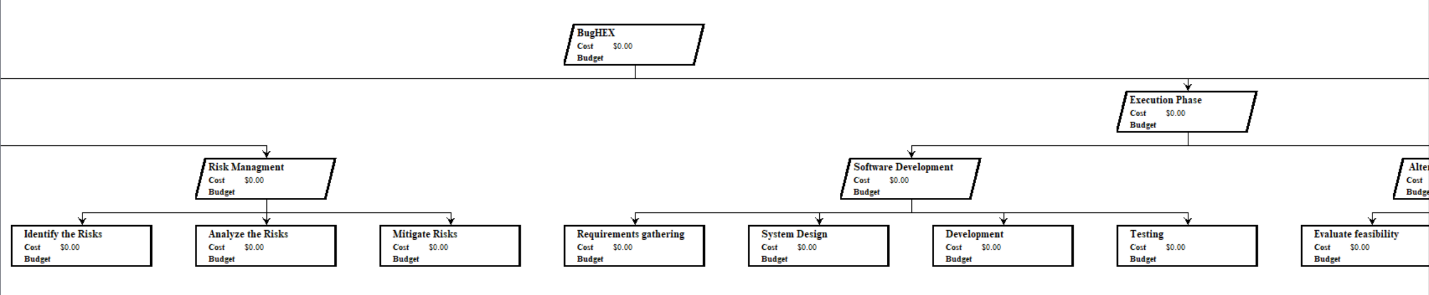
Figure(18)WBS 

Figure (19) WBS

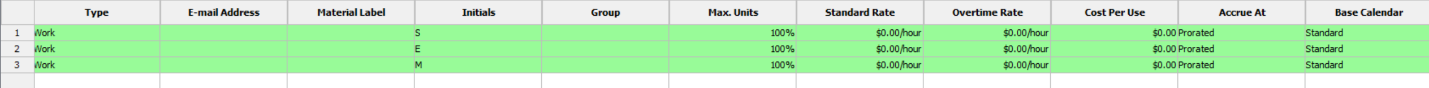
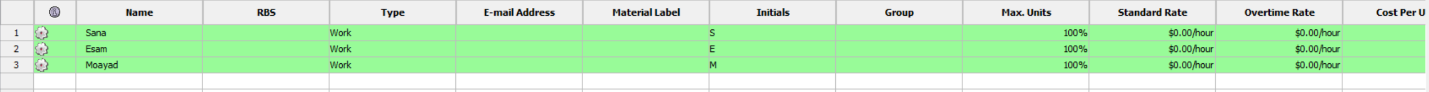


Figure (20) recourse tables

Essam's Table:

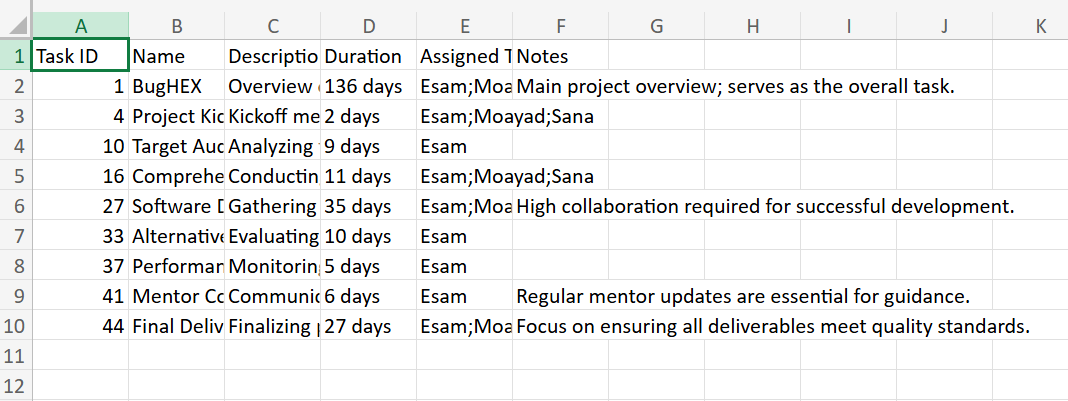


Figure (21) Essam's table

Sana's Table:

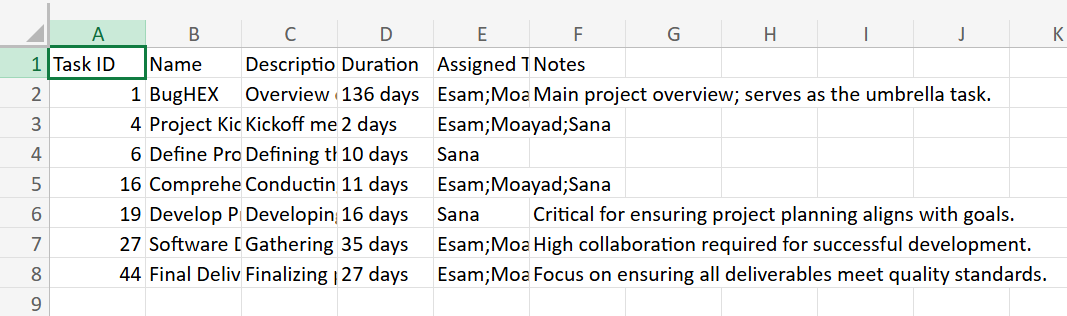


Figure (22) sana's table

Moayad's Table:

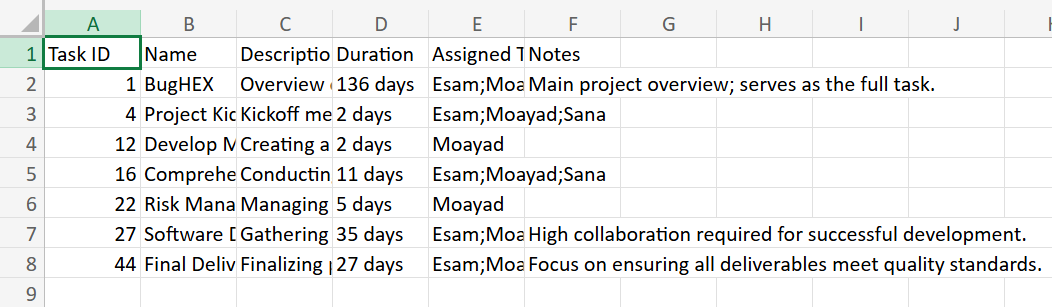


Figure (23) moayad's table

Use case diagram:

**1\_ Assembly Code Analysis:**

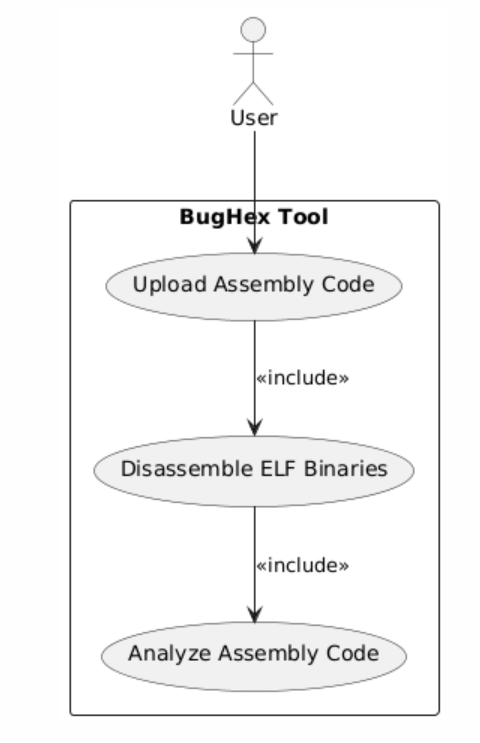


Figure (24) use case for assemble code analysis

**2\_Static Analysis:**

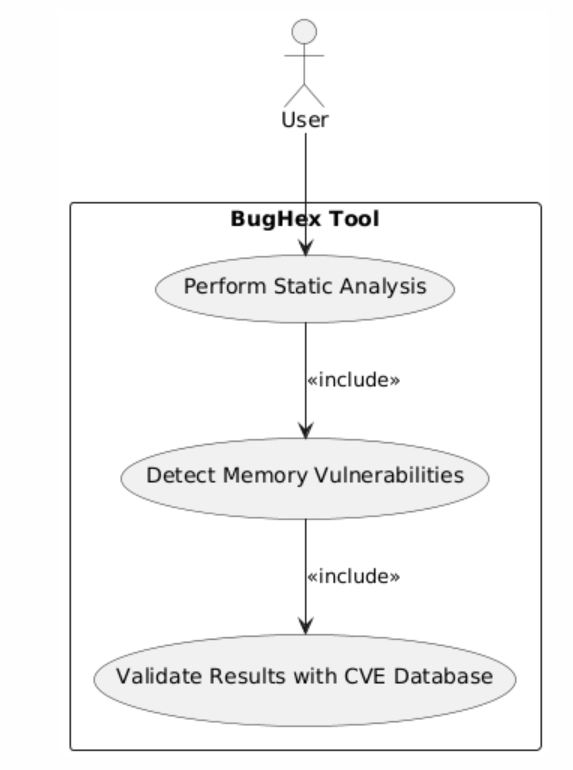


Figure (25) use case for static analysis

**3\_ Multi-assembly files types Support:**

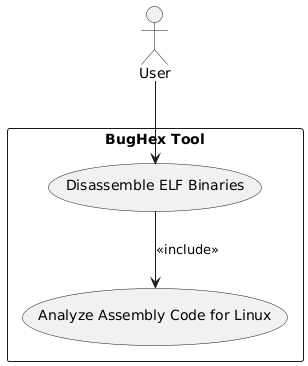


Figure (26) use case for multi assembly file type support

**4\_ Validation Using Known Vulnerabilities:**

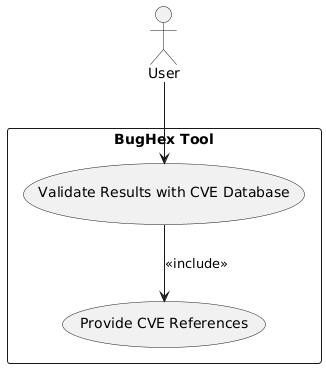


Figure (27) use case for validation of the CVE database

**5\_ Output Generation:**

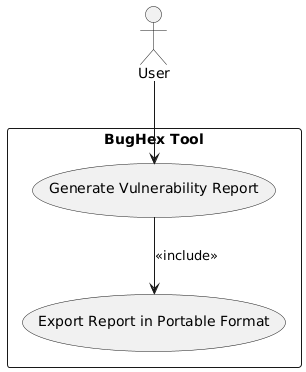


Figure (28) use case for report generation

**6\_ Optional Skipping of Steps:**

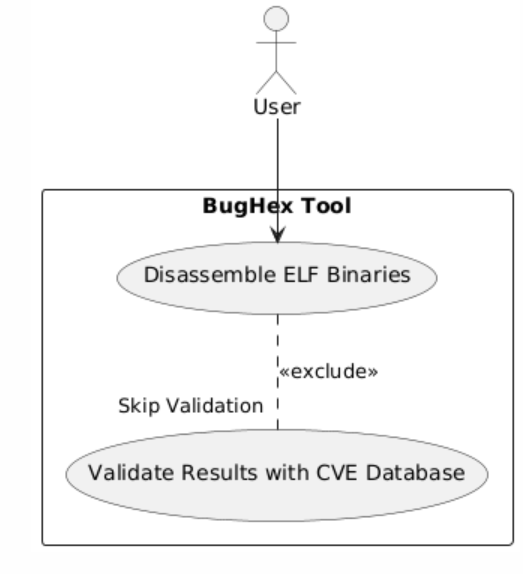


Figure (29) use case for optional kipping of the stage

sequence diagram:

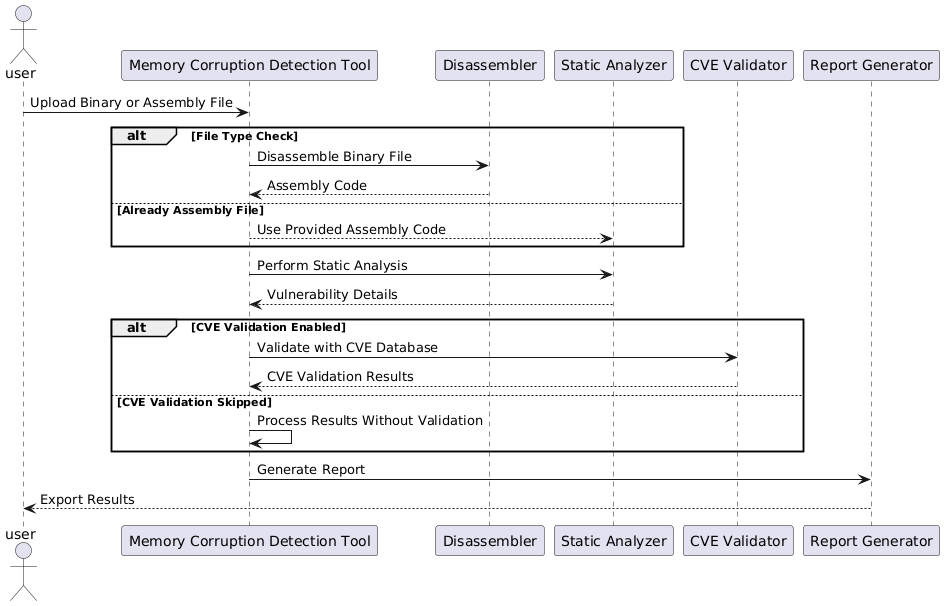


Figure (30) sequence diagram for the whole system

**Rationale Behind Choosing Vulnerability assessment:**

Our research concentrated on memory corruption vulnerability assessment in assembly code vulnerabilities since they represent the most severe and challenging security threats in the market. Memory corruption problems consisting of buffer overflows together with use-after-free and stack overflows present the greatest security hazards in software code which enables attackers to control systems and steal data and trigger severe security incidents. Low-level system operational knowledge is essential to understand such vulnerabilities which prove hard to track down with standard high-level security tools. The domains of digital forensics and web vulnerability in cybersecurity have already received extensive research support as well as multiple established tools but remain essential components. Low-level assembly code vulnerabilities in x86 and x64 architectures require additional attention because there is insufficient research and bridging solutions within this specific field. We focus on memory corruption vulnerability assessment to address an important industry gap because our specialized skills and background in IT and cybersecurity enable us to fulfill this need. Our expert skills match this domain perfectly and create an ideal situation to substantially enhance software security because this high-need area lacks effective specialized tools.

**Rationale Behind Choosing BugHex Project:**

Our team decided to create BugHex because various important conditions reflected our precise situation. Our time limitations together with our diverse skills as IT students from different majors shaped our decision to develop the BugHex tool. Our disparate skills united through this project which resulted in developing the design and implementation of the tool from multiple perspectives due to our software development and cybersecurity and systems analysis expertise. The project group became aware of the considerable security dangers posed by memory corruption vulnerabilities that remain a key vulnerability for software protection systems. These security weaknesses bring serious problems because attackers can breach system control pathways which means they need special attention before they cause harm. The serious security risks demanded an immediate solution which led to the creation of BugHex as an effective vulnerability identification instrument. Our academic setting enabled us to benefit from faculty mentorship together with technological support throughout the project development process and teamworking environment provided by fellow students. The tools we gathered enabled us to develop BugHex into an effective security tool for making valuable improvements in software protection systems. The development of our project established itself as a functional security answer while allowing us to translate classroom concepts into solving actual world security problems.

**BugHex: The Inspiration Behind Advanced Memory Corruption Detection and Patching:**

To address the problem of finding and patching the memory corruption vulnerabilities in low level assembly code for x86 and x64 architectures, there are a number of possible project ideas that use different technologies and approaches. However, it is in a prior penetration testing project when we first landed on the idea that led to the creation of BugHex, and it is this vision that is an integral part of our inspiration for these projects.

Thus one promising idea is to construct an improved static analysis tool that incorporates machine learning algorithms. It was meant to touch upon our prior successes with predictive analytics on the previous projects. The tool could improve the prediction and the detection of new vulnerabilities in assembly code by using historical data of known vulnerabilities. The tool could literally be integrated with machine learning to benefit from a continuous and modern improvement in detection capabilities with time, ensuring less false positives and negatives, and improved accuracy.

An alternative innovative solution is development of a dynamic analysis tool that provides real time feedback. This is an idea based on our previous experience with real-time systems monitoring attacks during a penetration test. It would run during the range of software system runtime, auditing for suspicious behaviors indicative of memory corruption. Immediate feedback and situation for detected problems would make it easy and efficient for developers to remove vulnerabilities.

With the strengths of static and dynamic analysis combined, a hybrid analysis system could also offer comprehensive security analysis. Such an approach is integrated and would give a robust defense mechanism, perusal of the code before its execution for apparent vulnerabilities and monitoring the software for any vulnerabilities at runtime.Moreover, aspects of positioning serial code within a localized simulation environment have their origin in our debugging experiments in earlier projects. Now, in this environment, the assembly code can run and be tested in the controlled manner similar to a confluence of the operating systems and hardware configurations. A tool that would allow you to test patches, you would use it to better understand how a piece of code behaves under another condition to avoid potential exploits.Lastly, given our penetration testing endeavors, the idea of vulnerability patching toolkit would aid with the remediation process that is not stream-lined by traditional patch management strategies. Automated scripts would be used in this toolkit to propose and execute fixes for the discovered vulnerabilities, seamlessly compatible with developers’ existing tools, and workflows for fast and effective patch deployment.Furthermore, each project idea addresses the difficult problem of the memory corruption in software systems in a practical manner, as well as drawing upon the challenges and successes in development that were observed on our previous penetration testing project, BugHex, to provide a variety of ways to both detect and respond to software memory corruption.

**Feasibility and Risk Assessment of the BugHex Project:**

Assess Feasibility  
1. Technical Feasibility: Our team consisting of computer science students with cybersecurity majors demonstrates solid abilities to address low-level software vulnerabilities because they combine knowledge with practical expertise. The combination of experts from different fields enables us to tackle technical memory corruption problems that occur in assembly code. Configuration along with testing for low-level programming and vulnerability assessment needs ongoing education and expert guidance from professionals knowledgeable in x86 and x64 architecture systems.

2. Operational Feasibility: Our project successfully implements operational feasibility because it depends only on free web-based resources that require minimal infrastructure investment or institutional backing. By remaining independent of university resources for development we can work more flexibly during both development cycles and deployment execution phases.

3. Economic Feasibility: The economic feasibility of our free web-based tools and resources produce a project that incurs no costs from its inception to date. The project remains affordable because of the implemented approach. The next stage of production and implementation requires systematic analysis of potential expenses to determine any needed funding sources.

4. Legal and Ethical Feasibility: The project follows entire computing laws and ethics by properly handling and disclosing security weaknesses in a responsible manner. Our project gives ethical concerns in cybersecurity the highest priority which directs us toward developing and managing software through secure ethical methodologies.

 Assess Suitability  
1. Alignment with Objectives: The BugHex project directly supports our main goal to develop a tool that provides significant improvements to identify and control memory corruption vulnerabilities. Our team can maximize both theoretical knowledge and practical experience to tackle an important security gap within the field.

2. Target Audience Needs: The core users of BugHex stem from penetration testers as well as cybersecurity researchers while developers who conduct system security evaluations round out the main audience. Relevant memory corruption capability within this project provides specialized features to fulfill significant user needs which typical standard security tools lack.

Risk Assessment  
1. Technical Risks: The main challenge lies in making BugHex accurately detect memory corruption problems while avoiding both false negative and false positive results. The system needs sophisticated analytical tools and detection software that will need repeated upgrades through testing and user evaluations.

2. Operational Risks: The implementation phase has not started so the tool needs evaluation regarding its capacity to integrate across various user environments and workflows. Beta testing combined with iterative development based on user feedback will be used as the main management strategy for this risk.

3. Financial Risks: There is a low financial risk at present because the project operates through free resources. The potential financial risks appear during development stages and wider implementations of the project. The project requires essential planning for potential expenses that we might need to obtain funding or partner with external organizations

References (Papers)

* Ieee.org. (2024). *IEEE Xplore Full-Text PDF:* [online] Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10496591 [Accessed 8 Dec. 2024].
* Ieee.org. (2024). *IEEE Xplore Full-Text PDF:* [online] Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10114930 [Accessed 8 Dec. 2024].
* <https://arxiv.org/pdf/2212.01254>
* Li, X., Feng, B., Li, G., Li, T. and He, M. (2021). A Vulnerability Detection System Based on Fusion of Assembly Code and Source Code. *Security and Communication Networks*, 2021, pp.1–11. doi:https://doi.org/10.1155/2021/9997641.
* Fang, M. and Hafiz, M. (2014). Discovering buffer overflow vulnerabilities in the wild. *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. doi:https://doi.org/10.1145/2652524.2652533.
* Sezer, E.C., Ning, P., Kil, C. and Xu, J. (2007). Memsherlock. doi:https://doi.org/10.1145/1315245.1315314.
* Yang, Q., Zhang, H. and Wang, X. (2023). Software Vulnerability Detection Based on Binary Intermediate Slicing. doi:https://doi.org/10.1109/qrs-c60940.2023.00087.
* <https://koreascience.kr/article/JAKO201914655594056.pdf>