1. Defining the Problem

Client

Business and home owners administrating WiFi LAN networks which include Internet of Things (IoT) devices.

Situation and Context

IoT devices are becoming increasingly popular as new developments in internet-connected technology makes both domestic and professional tasks easier and more effective. From internet connected fridges and thermostats to patient health monitoring systems in hospitals, IoT devices have substantial potential to revolutionise many aspects of our lives. However, IoT devices are also some of the most insecure devices available, a fact exacerbated by the fact many connect to the internet and are thus even easier to exploit. IoT network hardware devices have increasingly been used for both Distributed Denial of Service (DDoS) attacks, data theft and cryptocurrency mining[1].

Both cybersecurity researchers[2] and the American FBI[3] identify that "weak, guessable or hardcoded passwords" (OWASP) on internet connected devices are a major point of entry for IoT malware. Research by Kaspersky Labs in 2018[4] found that over 94% of malware variants targeting IoT devices attacked using brute force password guessing and predominantly used Telnet and SSH services.

Considering that there are expected to be more than 64 billion IoT devices worldwide by 2025[5], IoT security, especially the prevalence of weak passwords, should be a major consideration for network administrators. As such, tools need to be developed which allow network managers to protect their networks and devices from possible threats.

Client Needs

Clients thus need a system they can use to take a snapshot of IoT devices operating on their network and ensure they do not use weak passwords. This explicitly does not include changing or fixing weak passwords that are on the network, only notifying the user.

Developer Note:
This business case is likely to contain more technical details than is usual for user documentation. This is because the system's purpose is to provide technical information to the user regarding their network, meaning technical specifications and decisions which are usually only relevant to the developer's perspective are also key aspects of the end user's experience of the program. The system is also not designed for users without this technical knowledge because they risk damaging their systems and technologies.

---

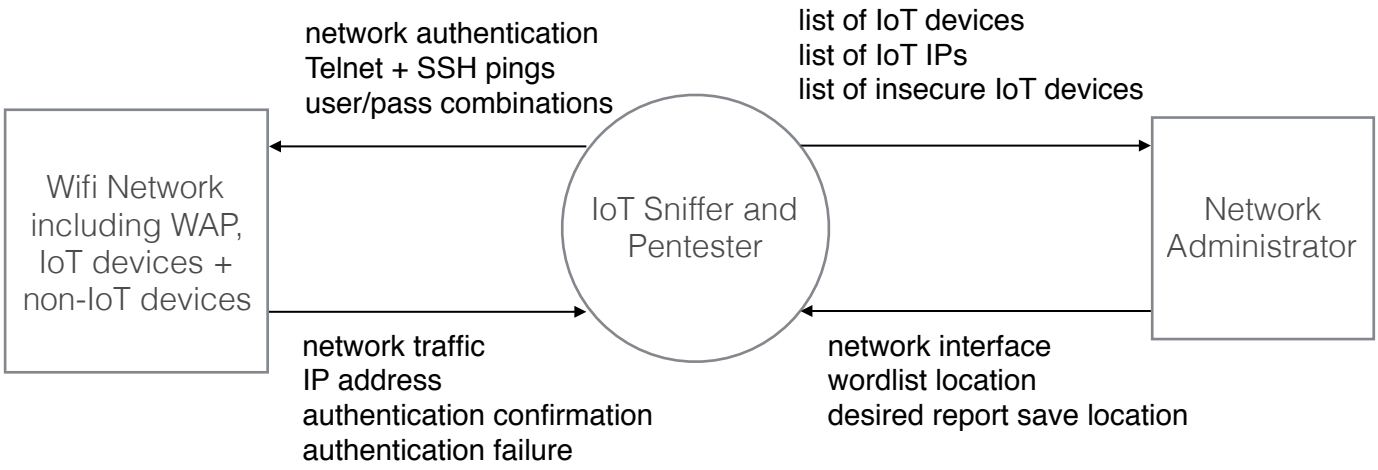
[1] Kaspersky Lab data accessible here

[2] Most notably the OWASP project, accessible here

[3] FBI Portland, accessible here

[4] Kaspersky Lab data accessible here

[5] Business Insider, IoT Analytics, Gartner and Intel research, accessible here



Developer Note:
Technically, the system interacts with the WAP, IoT devices and non-IoT devices separately. However, all of that communication goes through the WAP and the network. Because External entities shouldn't communicate with each other, I considered it more correct to treat the entire network as a single external entity

Essential Functionality (Core functionality required to address problem definition)

IoT Device Recognition: the program must be able to discover and recognise IoT devices operating on the network.

IoT Device contact: the program must be able to communicate with targeted IoT devices using both Telnet and SSH.

Root access brute forcing: the program must be able to test a list of usernames and passwords in order to try access the targeted device.

Process documentation: the program must be able to compile a report of discovered devices, their addresses and whether or not they could be accessed.

Command line interface: the program must be able to be used from the command line on non-GUI operating systems

standardisation: the program should operate within accepted cybersecurity industry standards and conventions.

Desired Functionality (Extra possible functionality that goes beyond problem definition)

Malware discovery: ideally, the program should be able to identify and kill malware it finds on unsecured IoT devices.

Honeypot: ideally, the program should be able to create its own honeypot on the network in order to track malware lurking in and around the network.

CLI + GUI: the program should have options to run it with either a GUI or CLI according to user choices and needs.

Identifying each device by brand or purpose: this would make users tracking down which of their devices are insecure easier.

Performance

Bandwidth: the program should be able to operate on the network without claiming too much of the network's bandwidth.

Speed: the program should be able to finish executing on a network within a reasonable timeframe (this will be different for each system depending on its size). The program should provide visual feedback to the user during its operation so the program does not simply appear unresponsive.

Compatibility

OS: the program should be operable across Linux, Windows and MacOS

network protocols: the program should be able to read traffic of varying protocols from each layer of the OSI model.

IoT firmware distributions: the program should be able to interact with a variety of IoT devices running varying operating systems.

Problem Definition

In summary, the problem context has four main parts:

1. IoT devices are fairly prevalent in LAN systems
2. Many of these systems have an unknown number of IoT devices operating on them.
3. Most IoT devices are incredibly insecure, mainly due to weak passwords.
4. IoT devices are a common target for malware

Accordingly, the user needs to be able to:

1. Know if and what IoT devices are operating on their network
2. Keep track of how many devices are operating on their network
3. Know which devices on their network have insecure authentication
4. Protect their IoT devices from malware

2. Design Specifications

General Description

This system utilises basic network sniffing functionality to discover and catalog IoT devices operating on a designated network. It performs a brute force attack on these identified devices in order to determine if any devices on the network have weak authentication. It then delivers all of this found information back to the user as a report. This allows network managers to determine how many IoT devices are operating on their network, which of them are insecure because of weak authentication and thus which are likely to be or have been targets of malware.

System Specifications

This system performs four main functions.

1. Network Sniffing[6]

The system listens to network traffic in order to identify active IP addresses.

2. IoT Device Discovery

The system analyses the protocols used in collected network traffic in order to determine which IP addresses are used by IoT devices.

This function, in tandem with the first, meets needs 1 and 2.

3. IoT Device Testing

The system uses Telnet and SSH to communicate with identified IoT devices and attempt to gain root access. This is achieved with a brute force dictionary attack using username and password combinations from the Mirai[7] and Gafgyt[8] botnets as well as the most common usernames and passwords from an anonymous but high profile dictionary list (containing 144 unique combinations initially linked to 8 233 specific hosts)[9]. These word lists and dictionaries were chosen because of their prevalence in the wild and use in high profile DDoS attacks.

This function meets needs 3 and 4 of the user.

4. Report Generation

Thus function compiles all of the data gathered by the first three functions into a user friendly report.

This ensures that the solutions for each of the user's needs in the other functions can be effectively communicated to the user.

See section 5 for interface design specifications

---

[6] It is possible this step will be replaced by an automated pinging of all IPs in range to see if they respond and that device identification will occur this way. Network sniffing is, however, preferable because it was a much smaller impact on network bandwidth.

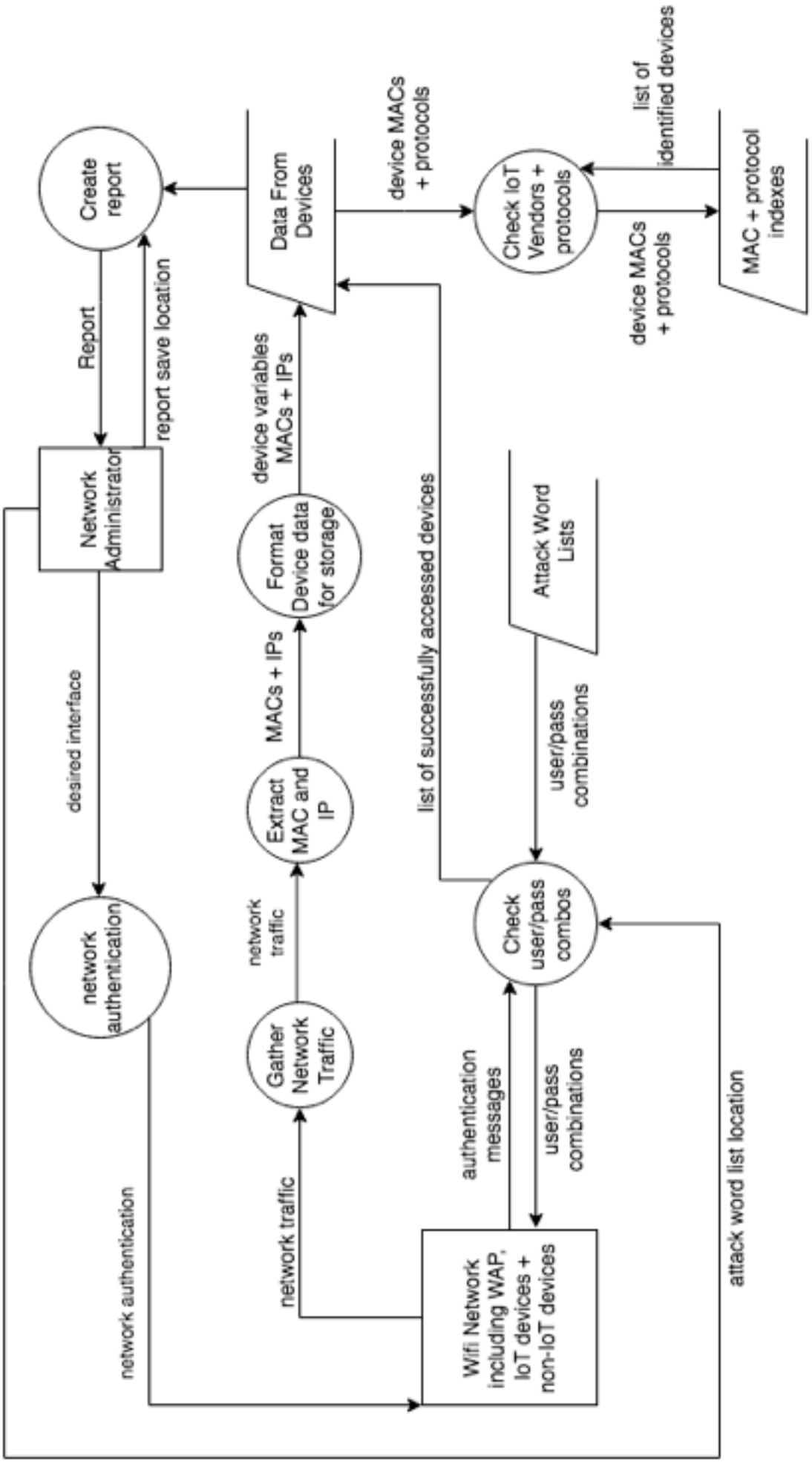
[7] published here

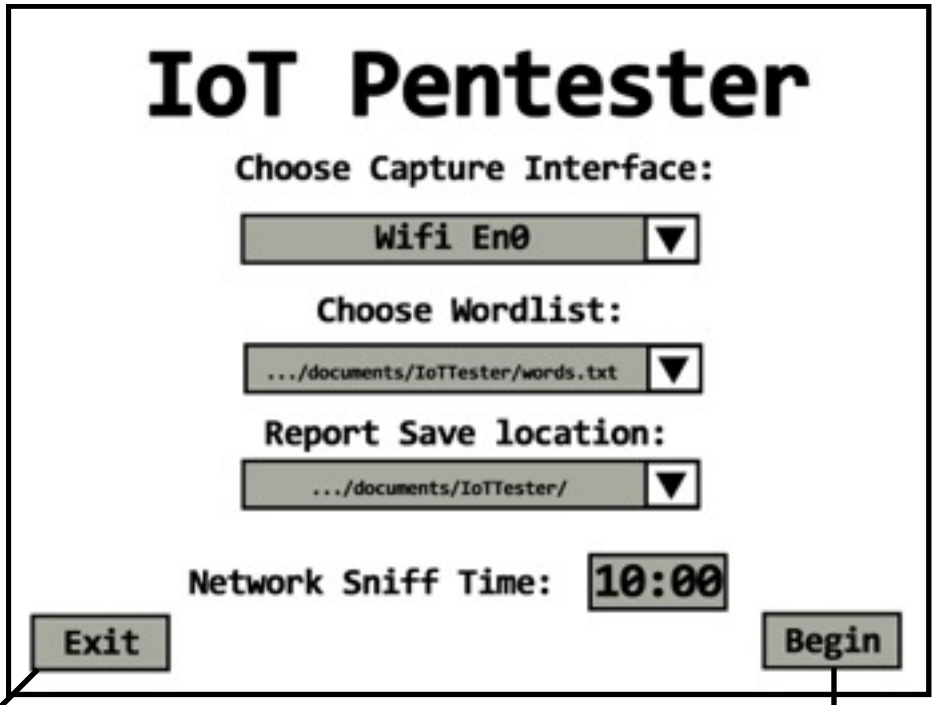
[8] published here

[9] published here

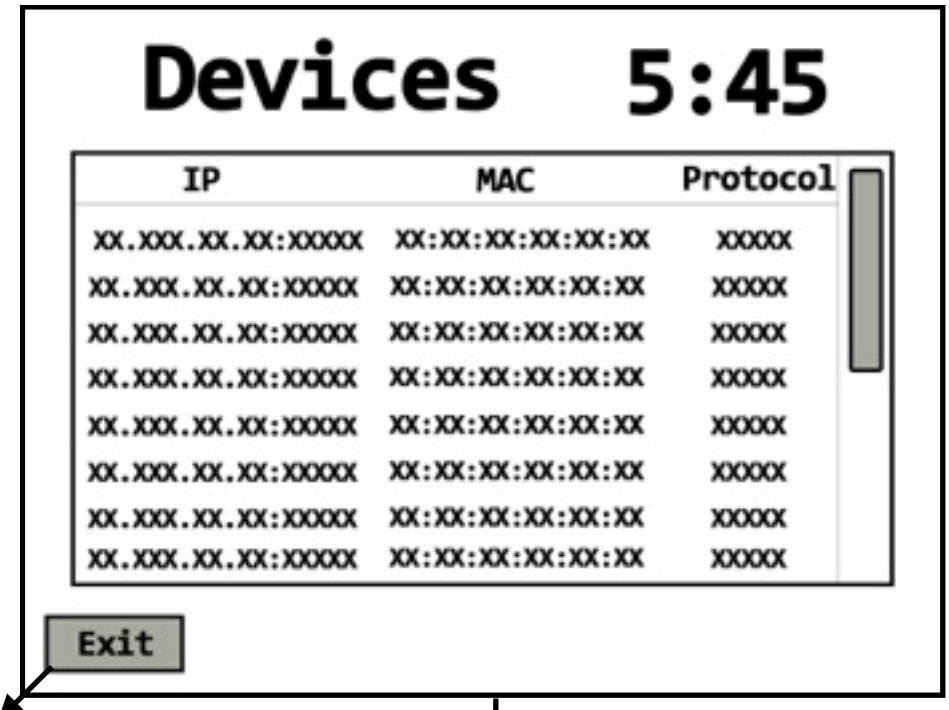
3. System Documentation

Function Description	Input	Process	Output
User Settings	Capture Interface Word list location report save location	save inputs	
Network Sniffing	network traffic	read MAC and IP addresses and protocols from ethernet frames create records for each device by IP in the ‘devices’, ‘IoTYN’, ‘InsecureYN’ and ‘login’ dictionaries	List of live IPs on network (‘devices’ dictionary)
IoT Device Discovery	List of live IPs on network (‘devices’ dictionary)	for each device: ‘IoTYN’ flag = 1 if MAC match known IoT vendors ‘IoTYN’ flag = 1if network protocol matched IoT standards and trends (eg MQTT)	List of IoT IPs (IoTYN dictionary)
IoT Device Testing	List of IoT IPs (IoTYN dictionary)	for each entry in IoTYN dictionary: read user/pass combinations from password dictionary file try to gain root access by entering each combination over telnet + SSH InsecureYN flag = 1 for successfully accessed devices record user/pass combination for successfully accessed devices in login dictionary	List of insecure devices (InsecureYN dictionary) user/passes for insecure devices (login dictionary)
Report Generation	list of IoT IPs (IoTYN dictionary) list of insecure devices (InsecureYN dictionary)	Create report printing IoT devices, their insecure flag and user/pass combination if found and save	Generated report for user


Create report
Data From Devices
device MACs + protocols
Check IoT Vendors + protocols
list of identified devices
MAC + protocol indexes
device MACs + protocols
Report
report save location
Network Administrator
device variables MACs + IPs
Format Device data for storage
Attack Word Lists
desired interface
MACs + IPs
Extract MAC and IP
list of successfully accessed devices
user/pass combinations
network authentication
network traffic
Check user/pass combos
Gather Network Traffic
network traffic
authentication messages
user/pass combinations
Wifi Network including WAP, IoT devices + non-IoT devices
network authentication
attack word list location

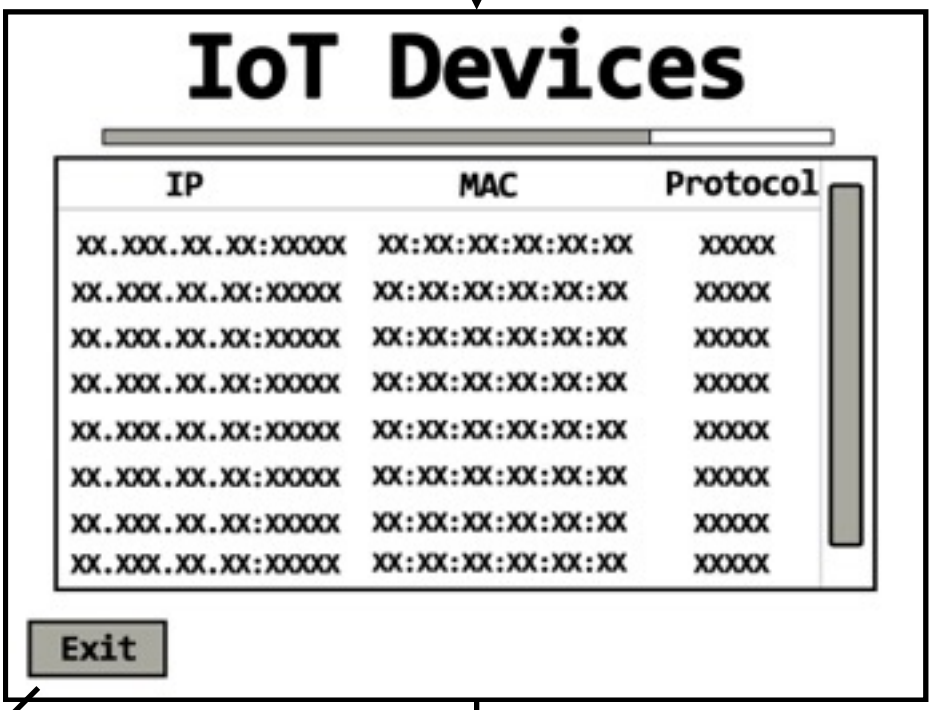


quit

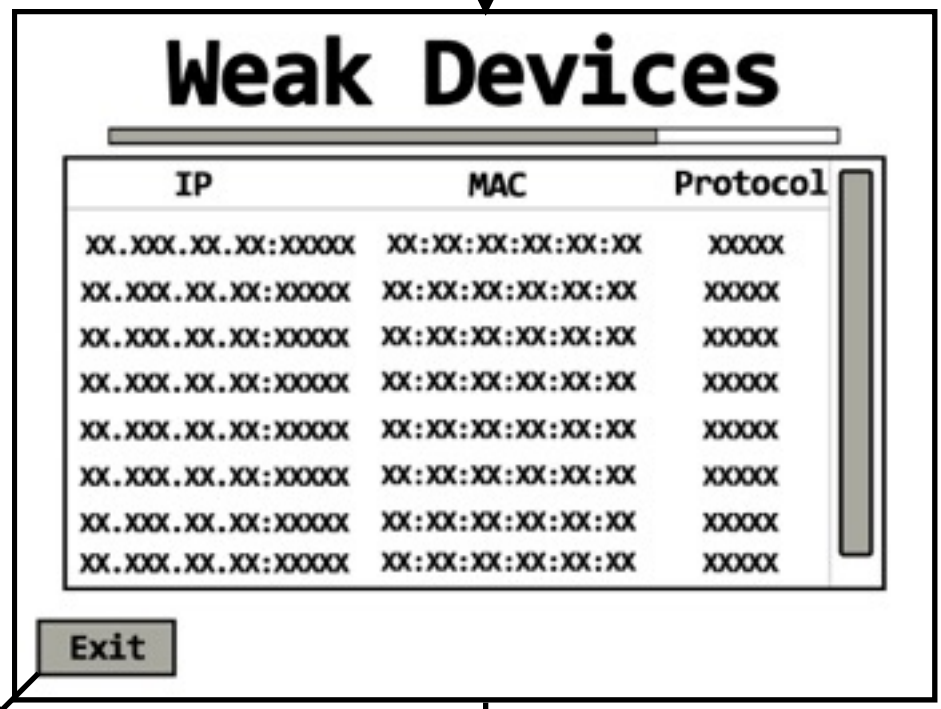


To main menu

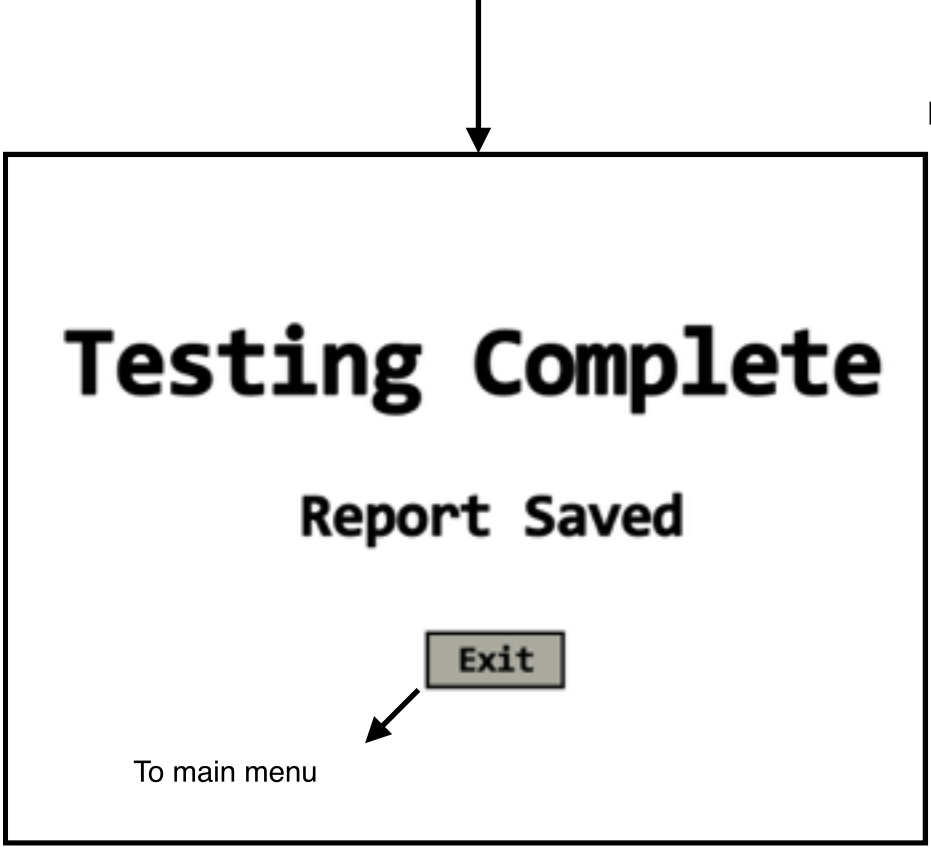
IoT Device Report **DD/MM/YY**



To main menu



To main menu



IP	MAC	Protocol	Insecure?	Username	Password
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX

4. Interface Design

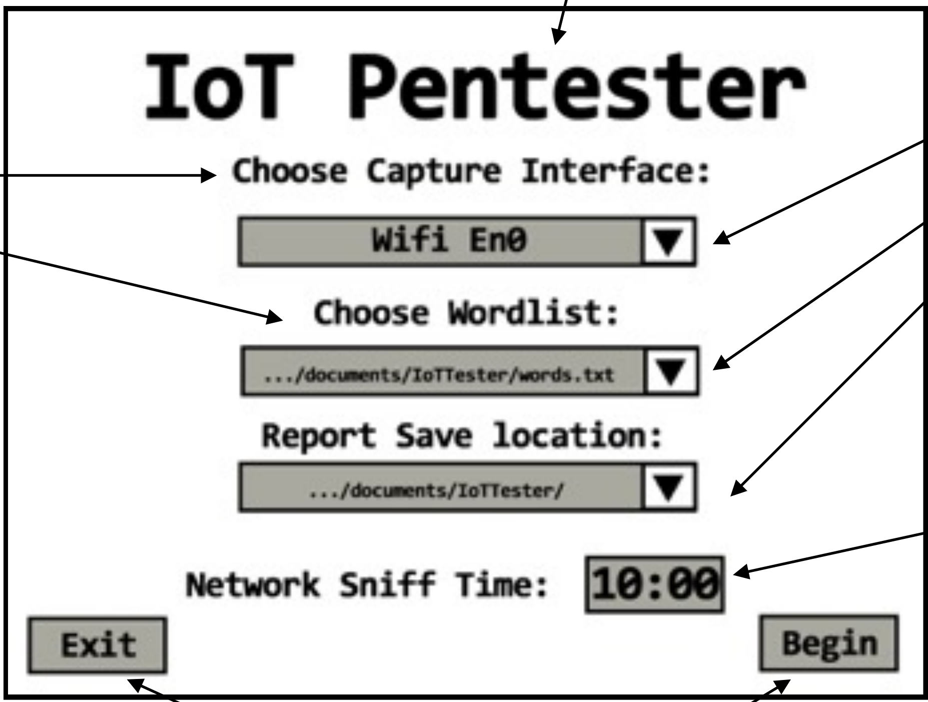
Applicability of UI to different displays
This program's UI should scale well to different displays because their is only limited information the user needs to read. The main concern is the readability of the grids in the three middle screens. However, the use of scroll bars means all required information can still be displayed without having to comprise on text sizing. While this may make the screen slightly less efficient to use at very small sizes, it should still be readable and the user will always have access to a report at the end of the process anyway should reading the screen ever become too difficult. The buttons, titles and text are all vector shapes, meaning they can scale up and down with ease. The lack of images or highly complex shapes also makes small screens easier to use. At very large sizes, the screens will have a lot of negative space. However, the ascetic of the program is not a very high concern and large screen also offer more a more comprehensive view of the grids.

This means scaling the UI to different displays should be easy and the system should operate on all displays.

consistent top-to-bottom flow of elements is intuitive and trains users for later screens

verbs in labels make it clear the user is expected to take an action

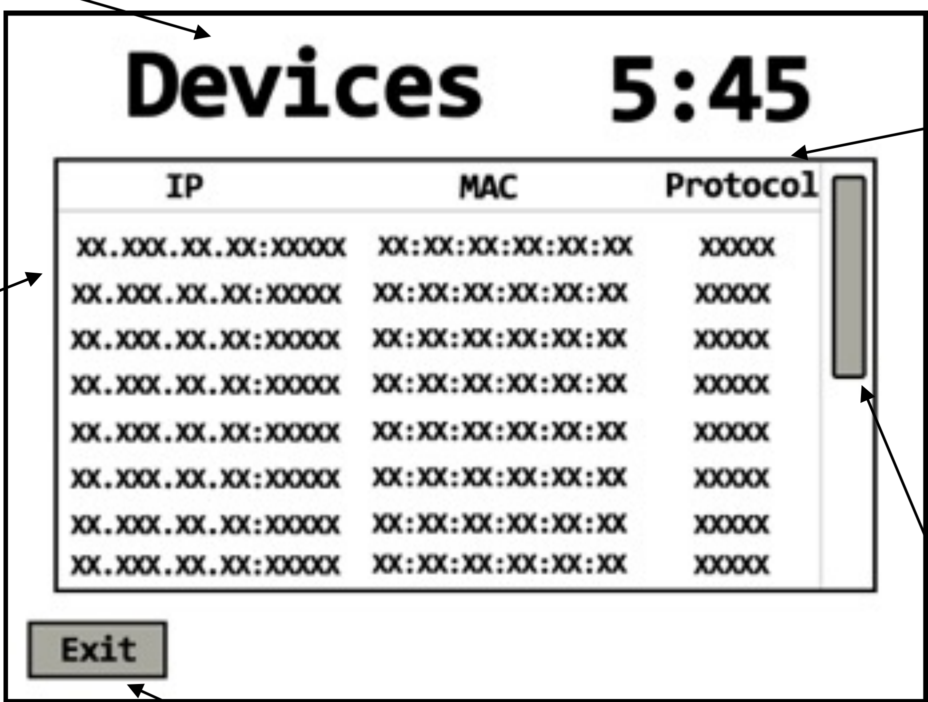
combination boxes ensure user picks from available options while also limiting the footprint of lists which could be quite long



text box with specific mask ensures valid input of a time in minutes and seconds

placing of buttons at the bottom of the screen makes them unobtrusive.

large title and timer means users can track the program's progress at a glance



labelled grid presents information logically and is consistent with other network tools, allowing user skills to be transferred across programs

list of devices which updates in real time ensure user feels like they are getting a response from the program. It also means cursory information can be gleaned from the system without having to wait for it to complete every step of its execution.

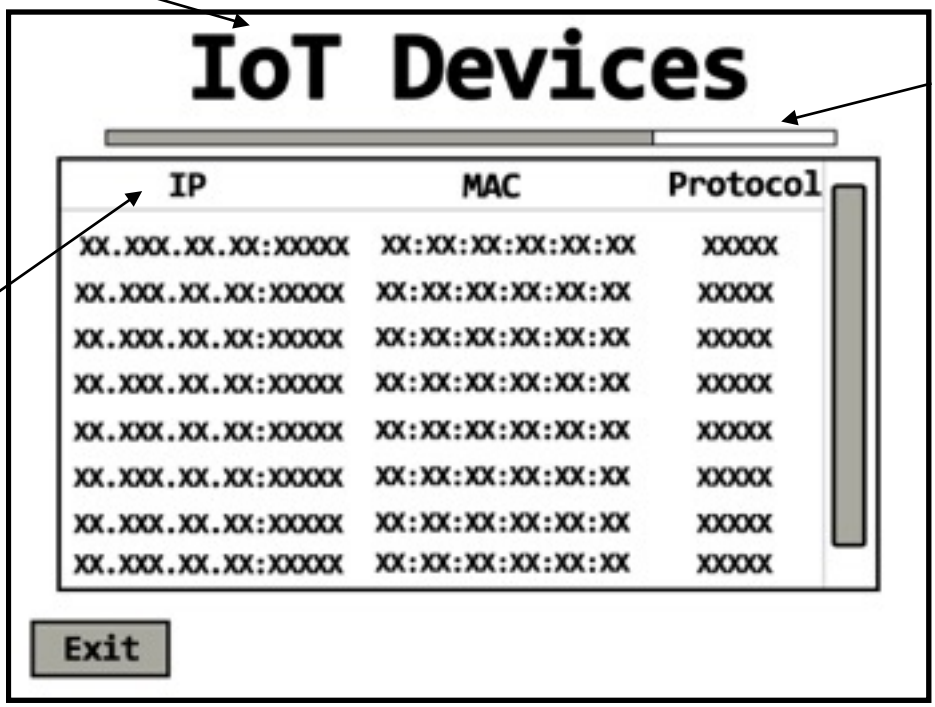
consistent button location makes use of the program easy and intuitive

scroll bar means list can expand indefinitely without sacrificing readability. This also gives users a sense of how much data the system has collected and what part of the list they are looking at

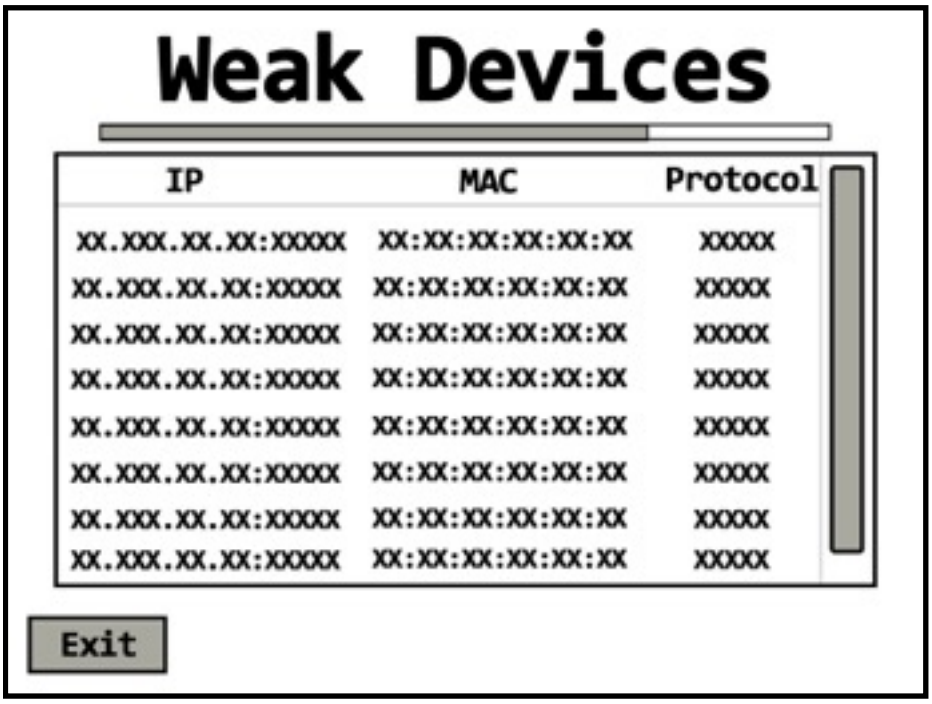
labels are consistent and match described functionality in system documentation

progress bar displays that the system is responsive by providing feedback. Users can also estimate at a glance how much longer this stage will take

grid is consistent with previous screens, meaning users have less to learn and can interpret program outputs quickly



general consistency with all previous steps



this layout is visually distinct from all preceding screens, makes it clear the program has finished

exit button's centring and size makes it a focal point for this screen. This emphasises that the program is done.

report is in the same visual style as the other screens, making it easier to read and interpret

Clear label makes purpose and role of data obvious

IoT Device Report DD/MM/YY

IP	MAC	Protocol	Insecure?	Username	Password
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX
XX.XXX.XX.XX:XXXXX	XX:XX:XX:XX:XX:XX	XXX	Y/N	XXXX	XXXX

grid formats data in a familiar way which uses many of the same fields as are present within the program

grid fields relate directly back to the main functions of the program, providing all information the user requires

lack of scroll bar or any other interactive elements means this report can be printed or displayed elsewhere with ease

5. Programming Language Selection

Overview of Sequential and Event Driven Programming

In a sequential paradigm, set pieces of code are executed line by line. This results in a linear process where the user responds to specific prompts from the program and has no access to other functionality. Thus, the user is subservient to the program and follows its instructions.

Event driven programming is the opposite. Rather than providing the user a specific course to follow, the computer defaults to a waiting state until an event triggers a specific handler function within the program. These events are generally either time based (so they execute after a certain amount of time has elapsed) or input based, where the program executes specific code based on some input from the user. In this way, the computer is subservient to the user, who has complete control over what the computer executes.

The Nature of this System

This system is predominantly sequential. However, the program's first screen is event driven beach it relies on user inputs to determine how the program will function. The main functions of network sniffing, IoT discovery, IoT testings and report generation are sequential because they are executed with a rigid and linear order of operations which does not change and does not require input from the user.

Language Selection: Python

I have opted to use Python to develop this system. This is because of the following four key reasons:

Readability
The fact this system contains predominantly sequential elements means that end user cannot just explore the interface and their options in order to gain an understanding of the system's functions and capabilities. In many event driven programs (such as games or even something like Adobe Photoshop), exploring menus and functionality is an effective way to learn what the program does and how it works. This level of interactivity is not available in a sequential system and thus this deeper level of understanding can mainly be gleaned through the user's exploration of the source code and documentation. Python's readability is crucial in this regard because it makes it easier for the end user to quickly look over source code and understand what the program is doing. The fact that Python is a high level language makes it closer to English and other natural languages, enhancing its readability. Python's adherence to the off-side rule ensures developers produce at least marginally clear code through consistent use of indentations throughout and across different programs.

Expansive Libraries
Python has an vast range of libraries, providing pre-scripted functionality. This assists Rapid Application Development by allowing a developer to quickly and efficiently produce a program. While this project is unlikely to be developed using a full RAD approach, the constant testing and redeveloping which occurs in the Agile approach (which will likely be used for this development) is also made easier by Python's extensive libraries. In particular, the socket and struct libraries make the development of this system much simpler. The socket library provides a low-level network interface for capturing network packets, while the functionality to interpret binary as strings provided by the struct library allows packets to be analysed. Without this provided functionality, the development of this system would be much more time intensive and inefficient because of the extra knowledge, skill and time required to produce equivalent code.

Interpreted
Python is an interpreted language, which provides benefits for RAD and debugging. The ability to quickly produce code and immediately run it without taking extra time for the code to compile is highly conducive to quick experimentation, which will likely be quite important to the development of this program. Python's interpretation also means the code should be executable on multiple platforms without requiring multiple binaries, making the program more flexible for use across a range of systems and networks.

Community Support
Python is also widely used by other prominent members of the cybersecurity community. For example, it is the standard language for penetration testers with the Australian Signals Directorate. This means it will be easy for other users of this system to incorporate their own their own functionality, either from tools they have created or found elsewhere. In addition, many mixed code environments have been developed in order to combine python with other languages. Thus, users should be able to incorporate code from other languages as well in order to tailor this system to their exact needs. Python's wide use also means that it is extensively documented and there are strong help communities available online for troubleshooting.

Why Not Lua?

Lua is another extremely popular language for the development of network cybersecurity tools and is used by both Nmap and Wireshark. However, there are a couple of reasons Lua seemed suboptimal.

I don't particularly need an embedded language
One of the appeals of using Lua is that it can be embedded in other code, offering a high-quality API for interfacing with other software like Nmap or Wireshark. However, this system does not need an imbedded language as it doesn't allow users to create their own programs within the system.

This system will use entirely original code
The fact that this system will not incorporate modules or functions by other members of the community means the program doesn't need to be in a language used for other similar tools. If anything, creating this system in a different language gives users a wider variety of tools they can tailor to their specific purpose. This is especially true because this system will not rival tools like Nmap or Wireshark in its power or functionality and is not designed to.

Error handling support is one of Lua's weaknesses
Lua has some documented issues with error handling support, which will be a crucial part of the development of this project. This system will be developed using an Agile approach to experiment with code and different functionality, which will require a robust system of error handling from the chosen language. This makes Lua unsuitable for the chosen approach.

6. Quality Assurance

The three key areas of quality assurance for this system will be modifiability, modularity and modifiability. These three were chosen as the key criteria for the system's quality because they determine the ability of other users or prospective developers to use the system under a range of conditions and with a variety of needs.

Modifiability
Modifiability is a measure of how easily and effectively the system can be changed or adapted to suit different user needs. Ensuring this system is highly modifiable guarantees that the system can be as effective as possible for as many users as possible. This is especially critical for a cybersecurity tool because many users are likely to have specific or nuanced needs when testing IoT devices. The fact that the user or client is not an explicit individual for the purposes of this system also makes the above points even more relevant.

Modifiability will predominantly be measured by the systems documentation because it provides users with the most knowledge about the system. This understanding should allow them to make specific and precise changes to the system in order to adapt it to their specific needs, without unintentionally changing anything else. The documentation will cover everything from the specific execution of specific liens of code to why modules where written how they are and descriptions of the system in its entirety. This provides a wealth of information not only on what the finished system does but the thought process and decisions behind the finished product. This provides the end user with all the information the developer had access to, best situating them to tailor the program.

Intrinsic documentation will be checked by asking the following two questions during each testing phase of Agile development:
- Is the style consistent throughout the source code?
- Are variables, functions and any other identifiers meaningfully labelled?

The following style guidelines will ensure the quality of intrinsic documentation:
- Code is indented as required by Python.
- Each of the programs four core functions will be separated by dotted comment lines.
- Selection and repetition control structures, variable and function definitions, and comments will all be separated from surrounding code by an extra carriage return either side.
- Variables, reserved words and operators will be coloured according to IDE style.
- Identifiers will be written in camel-case and begin with an abbreviation denoting their type.
- Nouns will be used to signify variables and other static value identifiers while verbs will denote a process or function.

Internal documentation will mainly consist of comments. Comment will describe:
- The operation and roles of each variable (upon definition)
- The operation and role of each function (at the beginning of its code)
- The operation and role of major control structures
- Specific design notes about lines which caused problems during development
- Specific design notes about lines which are highly unclear but critical to the program's operation (this is likely to be the highly technical lines produced by functions from the socket and struct libraries)

External documentation will contain:
- system models (context diagram, DFD, IPO diagram, storyboards, data dictionary)
- qualitative descriptions of the system's sequential actions
- a design commentary detailing major bugs and design decisions
- a change log detailing differences between major versions
- pseudocode

- test data and expected output

The effectiveness of this documentation will be checked through small scale surveys from individuals who haven't seen the system.

Modularity
Modularity is a measure of the ability to break the program down into individual units which operate independently. This has a tandem purpose to modifiability. By making the system as modular as possible, users or other prospective developers are most able to capitalise on the written code for use in their own projects without having to worry about undeclared variables, uncalled functions or missing operations.

Modularity will be assured through the targeted testing of individual modules independent of the rest of the source code. This process will likely occur incidentally while testing the program but will also be specifically performed at each testing stage of Agile development.

Each of the major functions and individual processes detailed in the IPO diagram earlier in this document should be able to function independently. For processes dependent on input from other processes, there will be an added focus on preventing errors and offering alternate methods of data input, whether read from a source file or entered manually.

Maintainability
Maintainability is a measure of how easy the software is to keep operating as well as how much it needs to be modified in order to work. This is important because IoT development is a dynamic and largely unstandardised industry which is thus likely to undergo significant change over the next couple of years. These changes are difficult to plan for and could render this system useless. The ability to easily maintain this system to keep up with changes in the industry ensures that the system is continually relevant and continues to meet the security needs of the user despite changes in the underlying technology. Thus, in this context, maintainability refers to the changes to the program required to keep it relevant within changing contexts to meet changing user needs.

Maintainability will be ensured by making all recurring values variables. This means that if values have to be changed at a later date, only the variable definition will need to be updated rather than having to trawl through the source code to find every reference. This is likely to affect data like the protocols checked for in network traffic and the word lists used for testing devices. These are both likely to change and be used at various points throughout the program. Making a central reference makes maintaining the system's functionality substantially easier.

However, Maintainability is also assisted by the management processes outlined for modifiability and modularity. A deep understanding of the system's purpose and operation empowers the user to make the changes they need to make while keeping in mind the goals of the system. Access to extra documentation such as diagrams and change logs also means these users can see what has been tried before and what does or doesn't work while trying to develop the system further themselves. A modular structure makes updates and patches easier to implement because the update developer can be sure they change all required references and processes without having to read through every line of source code looking for references to the value or process they want to change.