Assignment Guidance and Front Sheet

This sheet is to be populated by the Module Tutor, checked by the Programme Team, and uploaded to Moodle for students to fill in their ID and submit with their assessment.

Student ID or IDs for group work	**2136249**

Module Title & Code	WM140 Cyber Systems Architecture and Organisation
Module Owner	Amila Perera
Module Tutor	
Module Marker	Amila Perera
Assessment type	Coursework
Date Set	06/12/2021
Submission Date (excluding extensions)	07/03/2022 (refer to Tabula)
Marks return date (excluding extensions)	(Refer to Tabula)
Weighting of mark	100%

Assessment Detail	See below
Word Count	Section A Part I – There is a 600-word limit for this part. Section A Part II – There is a 600-word limit for this part. Section A Part III – There is a 600-word limit for this part. Section B Part I and Part II – There is no limit for this section. There is a 10% margin on the word count. Excessive length may be penalised, and the marker may ignore any material over the word limit. The word count includes footnotes and tables but excludes references and appendices.

Module learning outcomes (numbered)	1. Explain the relationship between the abstractions used to represent programs and data, and their concrete representation on real machines. 2. Explain the relationship between the key architectural components of a modern, multicore processor. 3. Evaluate code at the assembly language level to analyse cyber consequences from insecure patterns of code.	
Learning outcomes assessed in this assessment (numbered)	1. Explain the relationship between the abstractions used to represent programs and data, and their concrete representation on real machines. 2. Explain the relationship between the key architectural components of a modern, multicore processor. 3. Evaluate code at the assembly language level to analyse cyber consequences from insecure patterns of code.	
Marking guidelines	***Criteria***	***Mark***
	Section A	
	Part I: *Relevant discussion with appropriate justifications*	20
	Part II: *Relevant discussion with appropriate justifications*	20
	Part III: *Relevant discussion with appropriate justifications*	20
	Section B	
	Part I: *Clear Illustration of the step-by-step approach with appropriate justifications*	20
	Part II: *Relevant discussion with appropriate justifications*	10
	Presentation	
	Report structure and references	10
	TOTAL	**100%**
Submission guidance	You must use the Harvard referencing system. All submissions should be made in PDF format. Please follow further guidelines on Moodle	

Academic Guidance	*Academic guidance to be provided throughout the module.*
Resubmission details	The University policy is that students should be given the opportunity to remedy any failure at the earliest opportunity. What that “earliest opportunity” means in terms of timing and other arrangements is different depending on Programme (i.e. Undergraduate, Full Time Masters, Part Time Postgraduate, or Overseas). Students are advised to consult your Programme Team or intranet for clarity.
Late submission details	If work is submitted late, penalties will be applied at the rate of 5 marks per university working day after the due date, up to a maximum of 10 working days late. After this period the mark for the work will be reduced to 0 (which is the maximum penalty). “Late” means after the submission deadline time as well as the date – work submitted after the given time even on the same day is counted as 1 day late.

Contents

Section A 4
Section B 10

Figure 1 - Main Differences between x86 and x86_64 (Suraj Shende, 2021, p. 86) 5
Figure 2 - Vulnerable Snippet of Code 6
Figure 3 - Types of Buffer Overflow Protection 7
Figure 4 - Insecure functions and their secure equivalents (Tawil, 2012) 7
Figure 5 - Cache Side Channel Attack (Su and Zeng, 2021) 9
Figure 6 - Executing Binary without input parameters 10
Figure 7 - Run binary with incorrect key 10
Figure 8 - Loading binary into GDB 10
Figure 9 - List of the available functions inside the binary file 11
Figure 10 - Creating Breakpoint at Main() 11
Figure 11 - Running binary with invalid key 12
Figure 12 - discovery of valid key 13
Figure 13 - Output result of binary when given the correct key 13
Figure 14 - Example of a hashing algorithm 14
Figure 15 - hashing data using salt 14
Figure 16 - CSPRNGs available for popular programming languages (Secure Salted Password Hashing - How to do it Properly, no date) 15

Section A

i. A student argues that it is **more challenging** to perform a **stack-based buffer overflow attack** in a **x86_64** system than a **x86 system**. Discuss your opinion about this statement using an example.

Both x86 and x86_64 architecture come from the same ISA family set. The type of architecture defines the way a processor manages and executes instructions (yida, 2020). Main Differences are shown in Figure 1.

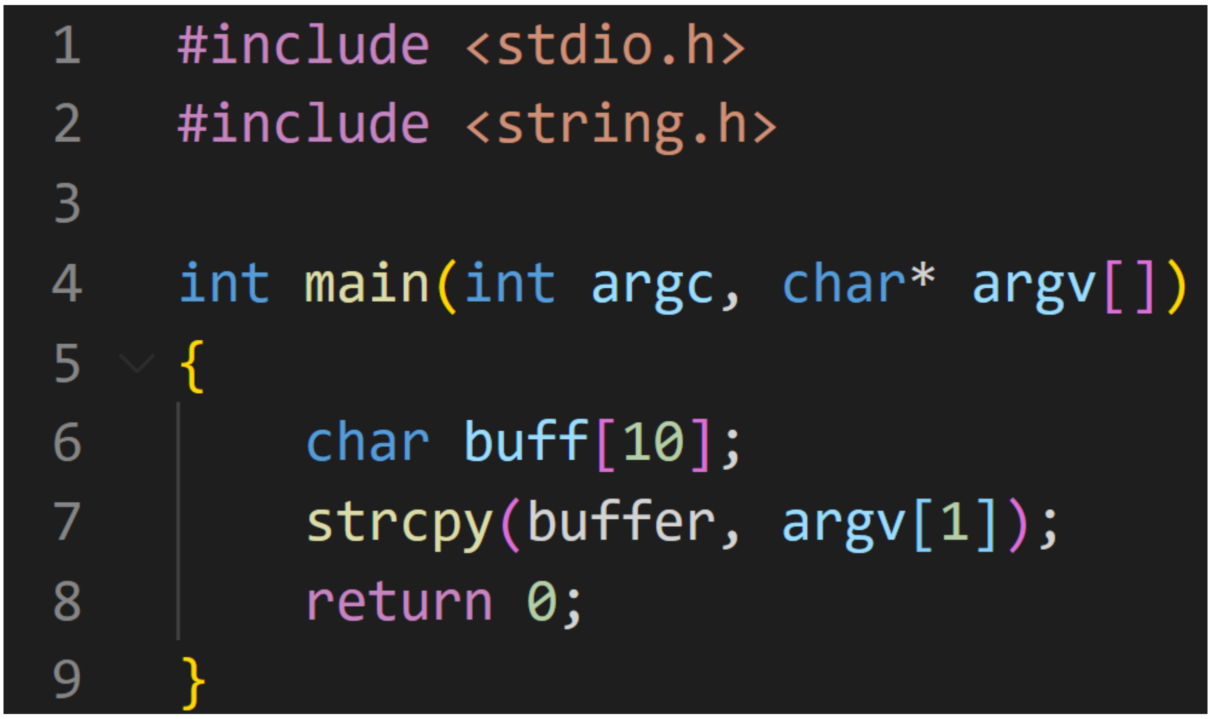
	x86	**x86_64**
Introduced On Year	1978	2000
Architecture	32-bit	64-bit
Registers, Memory, Data Bus	32-bit registers, data bus and memory.	64-bit registers, data bus and memory.
Addressable Memory	2^32= 4 GB	2^64 = 16 exabytes
Computation Power	Slower and less powerful	Faster processing
Features		

Figure 1 - Main Differences between x86 and x86_64 (Suraj Shende, 2021, p. 86)

As shown above, the size the CPU of the memory can access depends on the architecture, x86 is limited to 4 GB, whereas x64 can handle 8, 16 and 32 GB. However, having larger memory space does not imply more difficulty to perform a buffer overflow, because it mainly depends on the implementation and quality (Structure) of the code of a program. For example, using an unsafe function such as gets() is bad practice and should not be used on any of the architectures, this is because it passes all inputs from STDIN to the buffer without checking its size, enabling the possibility of a buffer overflow condition. Instead, developers should use the newest version known as fgets() which does the same job as gets() but requires the programmer to allocate a limit size buffer, eradicating the possibility of such an attack from happening (*CWE - CWE-242: Use of Inherently Dangerous Function (4.6)*, no date).

Similarly, the use of the operator **“>>”** is unsafe when reading into static arrays as it does not check the size of its input, allowing an attacker to send arbitrarily-size inputs and overflow the destination buffer regardless of the type of architecture.

Going back to the actual hardware, because the actual amount of physical and virtual memory that can be allocated is much lower than SIZE_T_MAX, in scenarios that might lead to overflow attacks will instead lead to memory allocation failure if using an x64 CPU architecture, which is much easier to detect. For example, an attacker can inject an arbitrary number of 32-byte structs in a buffer based on the length of the input, so trying to obtain 0x0800 001 will overflow a 32-bit SIZE_T but not a 64-bit one. One may argue that by increasing the input you can also overflow a 64-bit program, but it would be almost an unrealistic long input (*architecture - Security considerations of x86 vs x64*, no date).



```c
1    #include <stdio.h>
2    #include <string.h>
3
4    int main(int argc, char* argv[])
5    {
6        char buff[10];
7        strcpy(buffer, argv[1]);
8        return 0;
9    }
```

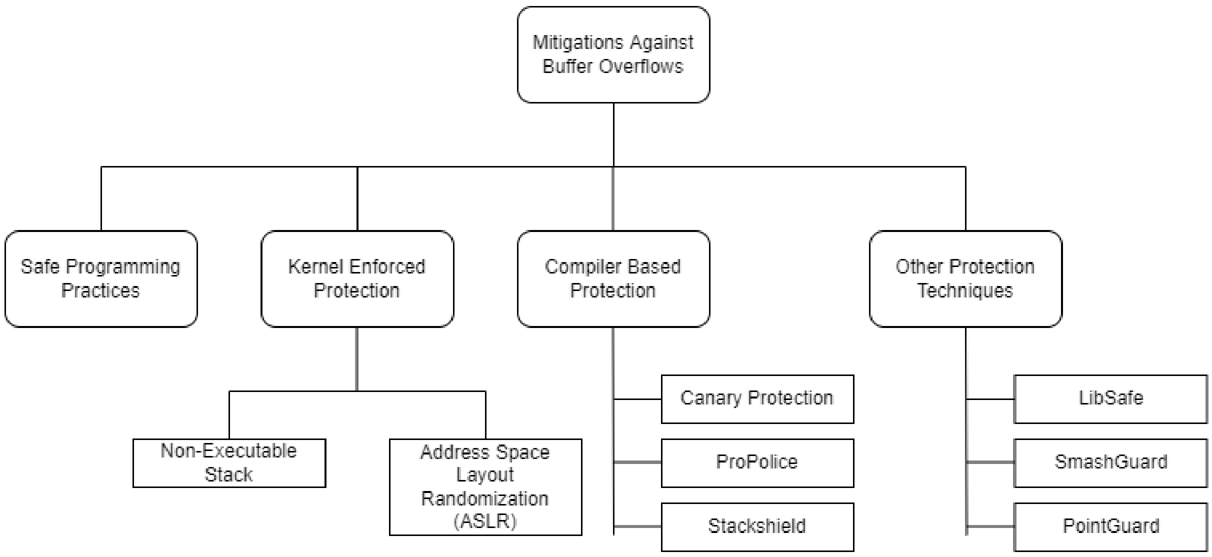
Figure 2 - Vulnerable Snippet of Code

Figure 2 shows a vulnerable example snippet of code written in C. We can see that this program is vulnerable to buffer overflow, as the program will copy the arguments from argv[1] into the buff[10] array using the strcpy() function. The problem begins at line 6 where "buff" represents an array of 10 bytes where buff[0] is the left and buff[9] is the right boundary. Assume the user inputs a size integer of 4 bytes the total buffer size of "buff[]" will be 10*4=40 bytes, this creates a buffer overflow attack where the remaining data is written into parts of memory that are not supposed to, an attacker can take advantage of this vulnerability to insert code into memory no matter what CPU architecture the user is using.

However, security measures have been implemented into newer architectures such as the x86_64 to prevent attacks like these from occurring such as non-executable stacks, address space layout randomization, stack canaries, control-flow integrity, and string function fortification.

ii. Discuss how modern computer systems mitigate buffer overflow attacks in multiple layers.

Buffer Overflows are still the leading cause of software vulnerabilities that can lead to critical security breaches if exploited by an attacker (CVE, no date). For this reason, new techniques and methods have been implemented into different layers to protect software applications and hardware from these types of attacks, shown in Figure 3: 1. safe programming practices, 2. Compiler based protection, 3. Kernel enforced protection and 4. Other additional techniques.


Mitigations Against Buffer Overflows
Safe Programming Practices
Kernel Enforced Protection
Compiler Based Protection
Other Protection Techniques
Non-Executable Stack
Address Space Layout Randomization (ASLR)
Canary Protection
ProPolice
Stackshield
LibSafe
SmashGuard
PointGuard
Figure 3 - Types of Buffer Overflow Protection

The most traditional approach to prevent and detect Buffer Overflows is to remove them from the source code itself. This approach requires the developer's constant attention to detail as well as huge technical knowledge with secure practices like buffer handling and memory management if working in programming languages like C. The approach is straightforward, replace unsafe functions such as strcpy(), strcat() that do not check the bounds of the target buffer with functions that perform the same functionality and consider the buffer size like strncpy() and strncat(). However, this extensive approach is not efficient as it requires manual work and does not guarantee the complete removal of the problem.

Insecure Function	**Replacement Function**
gets()	*fgets()*
strcpy()	*strncpy()*
strcat()	*strncat()*
sprint()	*snprintf()*

Figure 4 - Insecure functions and their secure equivalents (Tawil, 2012)

Modern Operating systems, incorporate Kernel-Enforced Protection, whereby the Kernel is allowed to alter the course of a program's execution. It does this by modifying the layout of a process' virtual memory address space and applying access controls to pages of memory which prevents the execution of arbitrary code (PaX, 2003b). Kernel protection includes Non-Executable Stack and Address Space Layout Randomization (ASLR).

Attacks such as the Stack Smashing - leverages the executable stack to overwrite return addresses to point to malicious code (technopedia, no date), illustrates the need to make some parts of memory as non-executable (NX) in particular the stack and heap areas, as injected code cannot be executed it minimises the severity of such vulnerabilities (PaX, 2003a), it is achieved by implementing three different functions R (Read), W (Write), X (Execute) and forcing the stack to only be on one those states at a time. This type of technique tries to eradicate buffer overflows involving minimal cost and its effectiveness against intended attacks. This is implemented in Linux using hardware features such as the NX (non-execute) bit to tag memory regions as non-executable. Windows OS uses DEP (Data Execution Prevention) which tries to prevent data execution via malicious code from memory locations reserved for the OS and authorised programs. However, this technique does not prevent the overwrite of any pointer stored in the memory region and others that do not involve code injection such as the return-to-libc attack which directs the program control to code located in shared libraries which cannot be prevented using a non-Executable stack.

Address Layout Randomisation (ASLR) assumes that exploits techniques rely on the memory layout of the program being targeted, its purpose is to create a level of randomness during the execution of a binary. This means that binary mapping, dynamic library linking, and stack memory regions get randomised before execution, which decreases the predictability of the memory layout and therefore reduces the likelihood of a successful exploit. This is because when the program detects it has been directed to an incorrect address space, the program tries to terminate its execution thereby stopping the attack from happening.

There are other protection features implemented by the compiler that takes a different approach, where the compiler has complete control over the structure of the binary so it can make appropriate changes to the stack layout. Techniques include StackGuard, ProPolice and StackShield. For example, StackGuard is an extension that proposes the detection of such attacks by arbitrarily placing a word-size canary() to detect modified data by buffer overflows and try to mitigate it automatically. However, a buffer overflow attack can still be done by overwriting local variables or functions instead of the canary value as it only protects the linked information in the stack.

In conclusion, buffer overflows can be mitigated using a combination of the techniques explained, although exploitation techniques will always be much more creative and capable of overcoming existing method of protection.

iii. Explain how side-channel attacks via CPU cache could affect the hardware security of computer.

Cache-side channel attacks utilise side-channel information like power, heat, and time to derive confidential information such as keys used in cryptographic systems. Today there exits software-based side-channel attacks that can exploit features in modern CPUs, without the need of physical access, or even direct contact to memory as they are conducted through legitimate software functions, making this attack one of the most serious threats for modern systems. Potential implications include the decryption of secret keys used in cryptographic operations such as the AES algorithm.

Currently, there are two cache-based side-channel attacks: access and time-driven attacks. In an access driven attack, the attacker has control over one or multiple spy processes that share the cache with the victim's process. Due to the cache sharing, the victim may be able to avoid the spy process' cache lines, when accessing critical cache lines, however, an attacker can measure the access time of the cache lines and figure out which are evicted by the victim process. By doing this behaviour analysis the victim process may leak enough information for the attacker to infer restricted information (Kong *et al.*, 2009).

Time-driven attacks have two approaches, one of them is referred to as miss-based attack which is based on observation where the attacker observers if the victim uses one of the attacker-evicted cache lines, which causes the victim to take a longer execution time cycle for a security-critical operation which is then leveraged by the attacker to gain more information about the victim, an example of this approach is the evict-and-time attack – if the victim uses the evicted data a cache miss occurs, therefore the execution time for the security operation increases e.g. block encryption in AES, and this can be observed by the attacker. Another approach is known as hit-based attack where the attacker observes if a victim is reusing memory lines fetched by its previous memory accesses to see whether the victim has reduced its security operation runtime due to hits in the victim's own memory accesses which can be useful for an attacker as they can implement attacks such as cache collision attack (He and Lee, 2017). The main objective of this attack is to measure the time for cache access and relate the timing values to the information being processed, as the cache access time can only be different in two cases: 1. When the requested data by the CPU is available in the cache (cache-hit) or when the cache does not have the data available and request it from the main memory (cache-miss) (*Chapter 8 - Side-Channel Attacks | Elsevier Enhanced Reader*, no date).

The harms of cache side-channel attacks are that the attacker can use this channel to secretly transmit sensitive information, posing threats to the system. For example, by exploiting this vulnerability an attacker can disclose sensitive information such as privacy data, execute arbitrary code or even perform a denial-of-service attack against the CPU. Cache side attacks can be also performed on different VMs, across multiple CPUs and platforms. Due to the unpredictability of the cache state, this type of sensitive information leakage is difficult to capture through pattern or feature detection, so it is concealed.

Cache side channel attacks can also be used to serve as another malicious operation, for example gaining access to a system by passing malicious code through the side channel. For example, ExSpectre uses a side channel to achieve code execution, as shown in figure 5. The attacker first teaches the branch predictor through trigger codes, so the next time the program encounters the same branch, it will take it to the predicted executable code. The red arrow indicates the execution of the misprediction, where malicious code gadgets are hidden, when the CPU realises the misprediction, it rolls back and executes the actual branch. However, the malicious instructions had already been executed, and the execution result of the instruction is transferred to the attacker through a side-channel attack on a predefined array. This means that the rollback does not eliminate the influence on the cache (Su and Zeng, 2021).

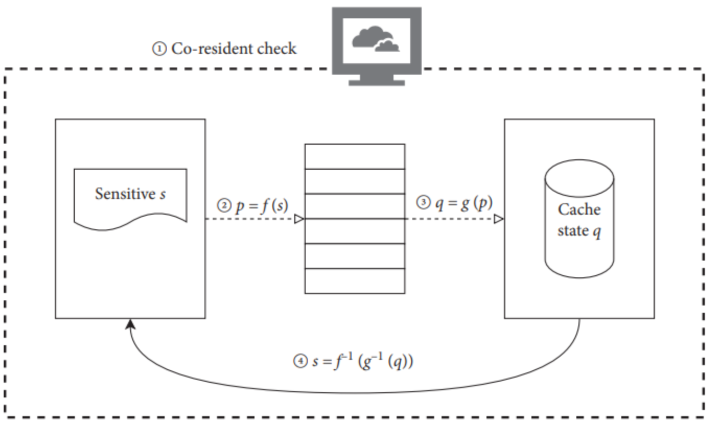


Figure 2: Workflow of cache-based side-channel attacks.

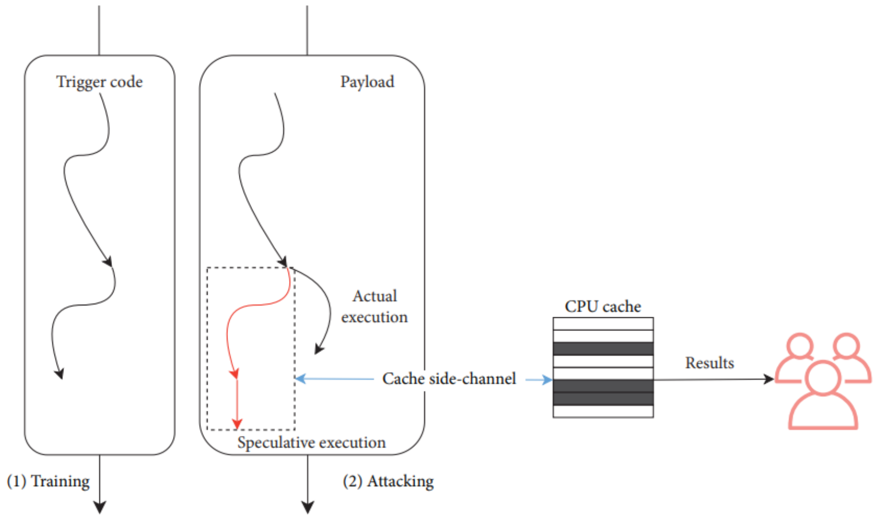


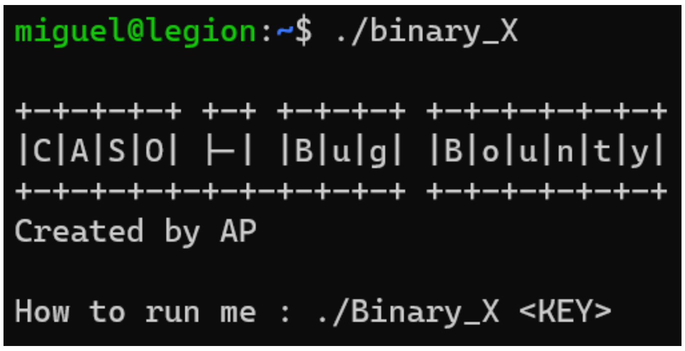
Figure 3: Leak malicious results of ExSpectre through cache-based side-channel attack.

Figure 5 - Cache Side Channel Attack (Su and Zeng, 2021)

Section B

i. You have been given a binary file that expects the correct passphrase as the user input to display the "Access Granted" message. Explain how you could use gdb to trace through the code and bypass passphrase validation during the runtime.

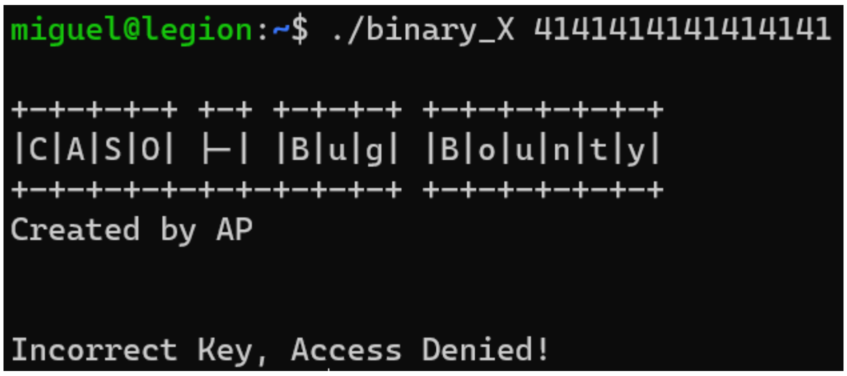
Running the program for the first time, give us instructions on how to run the program in this case the program will only continue executing the rest of the code only if the input given contains a key number, otherwise, the program ends, shown in figure 6.



```
miguel@legion:~$ ./binary_X

+-+-+-+-+ +-+ +-+-+-+ +-+-+-+-+-+-+
|C|A|S|O| |-| |B|u|g| |B|o|u|n|t|y|
+-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+
Created by AP

How to run me : ./Binary_X <KEY>
```

Figure 6 - Executing Binary without input parameters

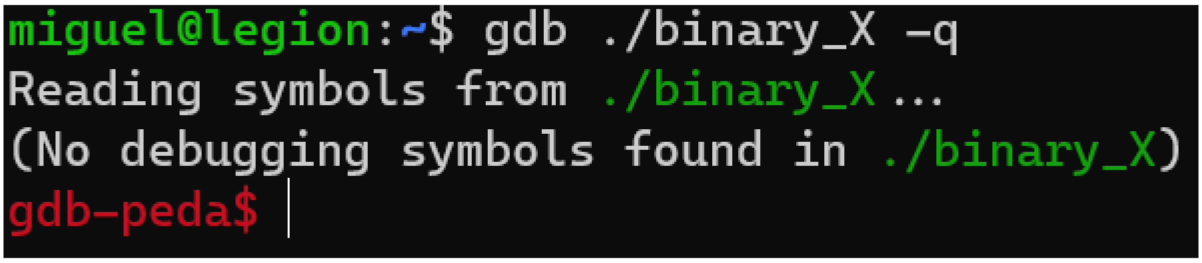
We would like to know how the program would behave when an incorrect key is given. In this case, I ran the program as instructed with a random input parameter, shown in Figure 7. As a result, we get an output message saying, "Access Denied".



```
miguel@legion:~$ ./binary_X 4141414141414141

+-+-+-+-+ +-+ +-+-+-+ +-+-+-+-+-+-+
|C|A|S|O| |-| |B|u|g| |B|o|u|n|t|y|
+-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+
Created by AP


Incorrect Key, Access Denied!
```

Figure 7 - Run binary with incorrect key

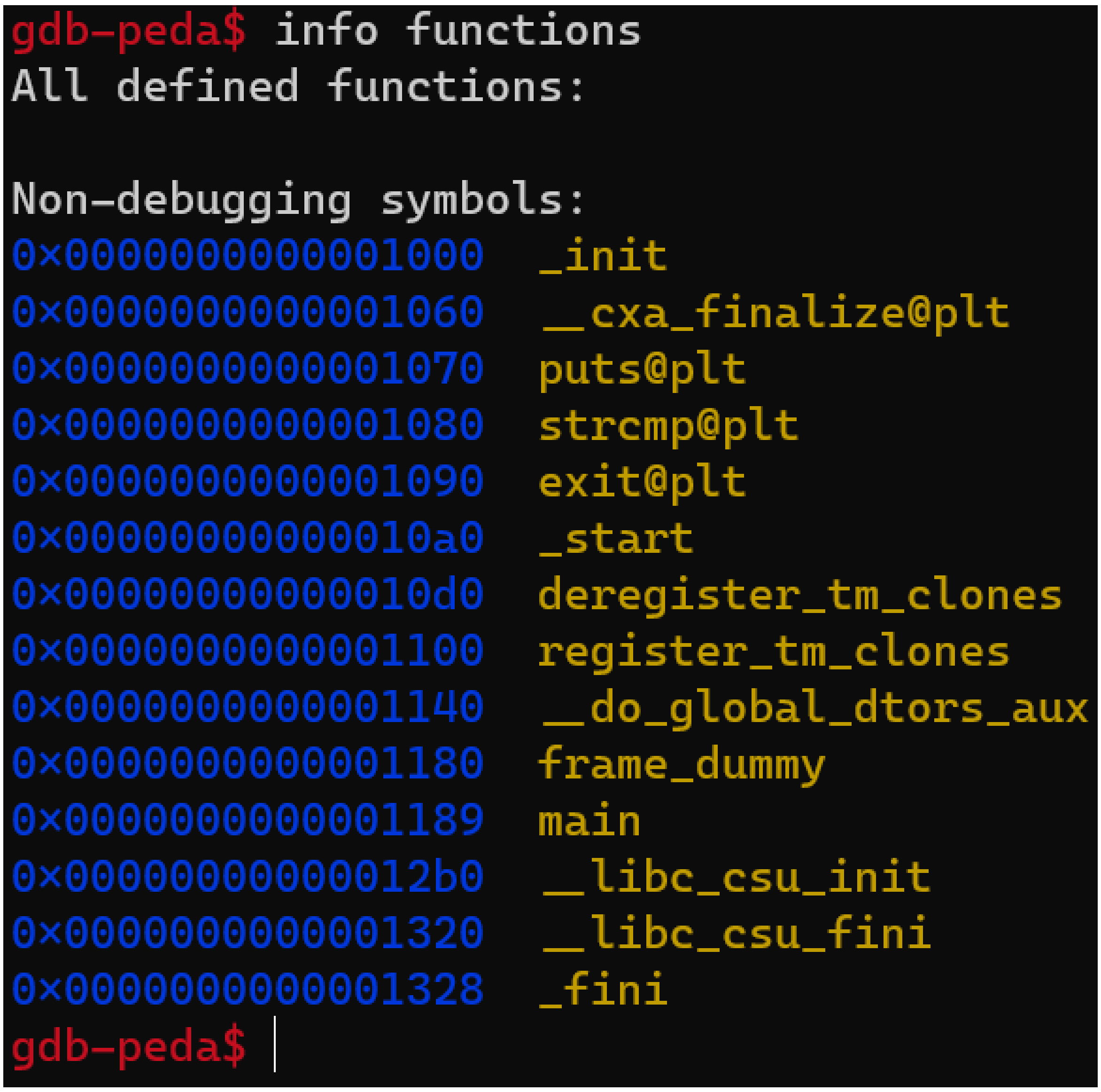
Of course, what we want is to gain access, so to do this we will have to debug the program using a tool known as GDB. The first step that needs to be taken is to load the binary file into GDB, as shown in figure 8, the -q flag is not needed but there to minimise the output.



```
miguel@legion:~$ gdb ./binary_X -q
Reading symbols from ./binary_X...
(No debugging symbols found in ./binary_X)
gdb-peda$
```

Figure 8 - Loading binary into GDB

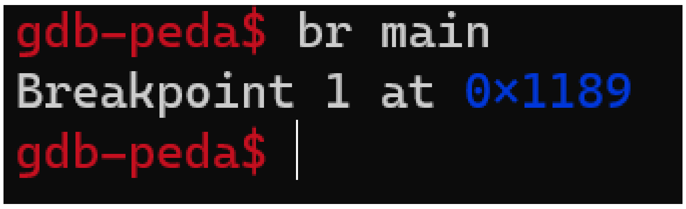
After the initial step is taken, we can continue by looking at all the available functions in the program that we can use to step into code and therefore figure out the correct passphrase, this is done by using the command shown in Figure 9.



```
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001060  __cxa_finalize@plt
0x0000000000001070  puts@plt
0x0000000000001080  strcmp@plt
0x0000000000001090  exit@plt
0x00000000000010a0  _start
0x00000000000010d0  deregister_tm_clones
0x0000000000001100  register_tm_clones
0x0000000000001140  __do_global_dtors_aux
0x0000000000001180  frame_dummy
0x0000000000001189  main
0x00000000000012b0  __libc_csu_init
0x0000000000001320  __libc_csu_fini
0x0000000000001328  _fini
gdb-peda$
```

Figure 9 - List of the available functions inside the binary file

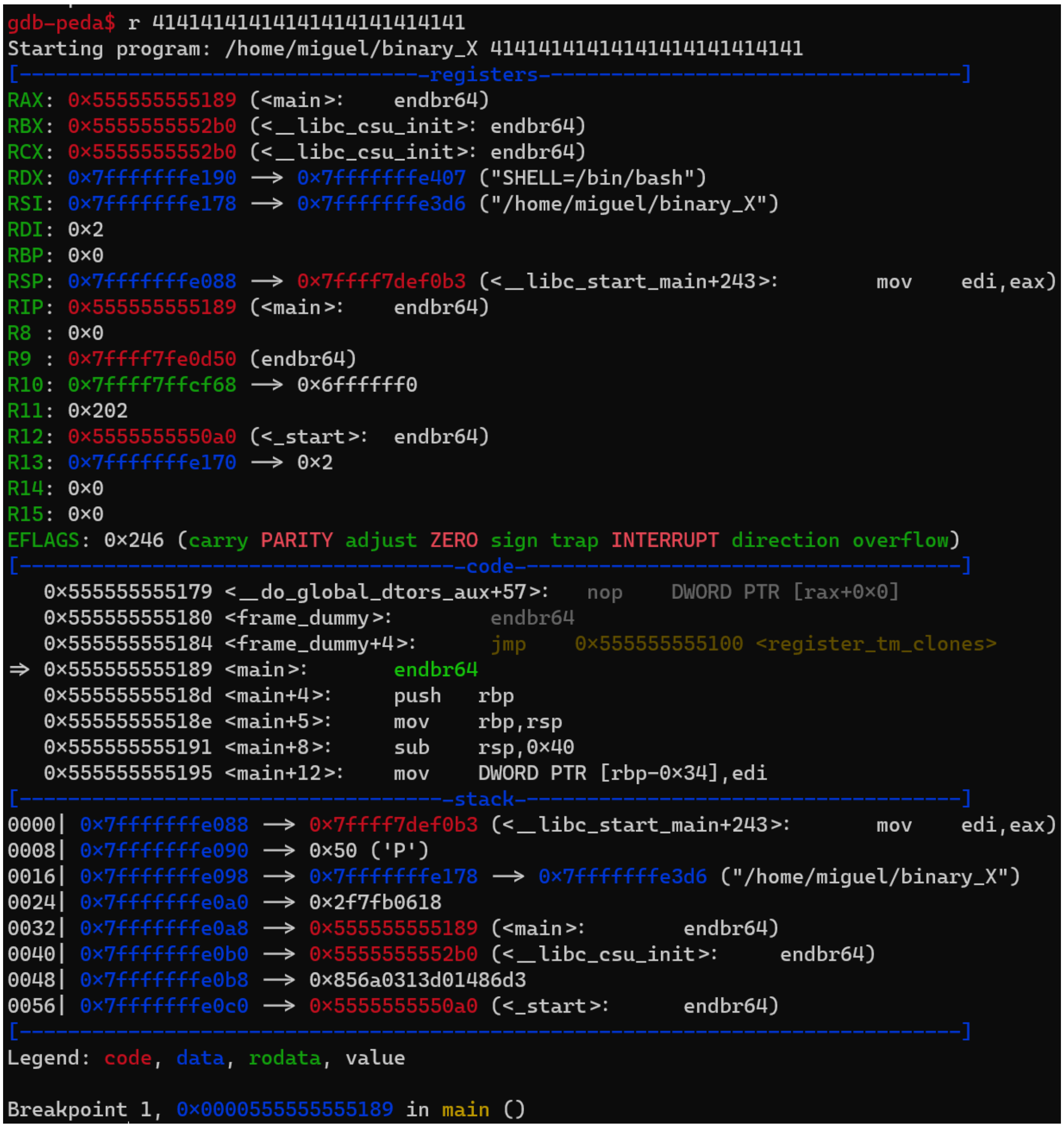
Once we know all the available functions available to use, we can handpick where we would like to create a breakpoint – a mark point that tells GDB where to stop the program from running. In this case I have decided to break at the main function, so we step through the whole program, shown in Figure 10.



```
gdb-peda$ br main
Breakpoint 1 at 0x1189
gdb-peda$
```

Figure 10 - Creating Breakpoint at Main()

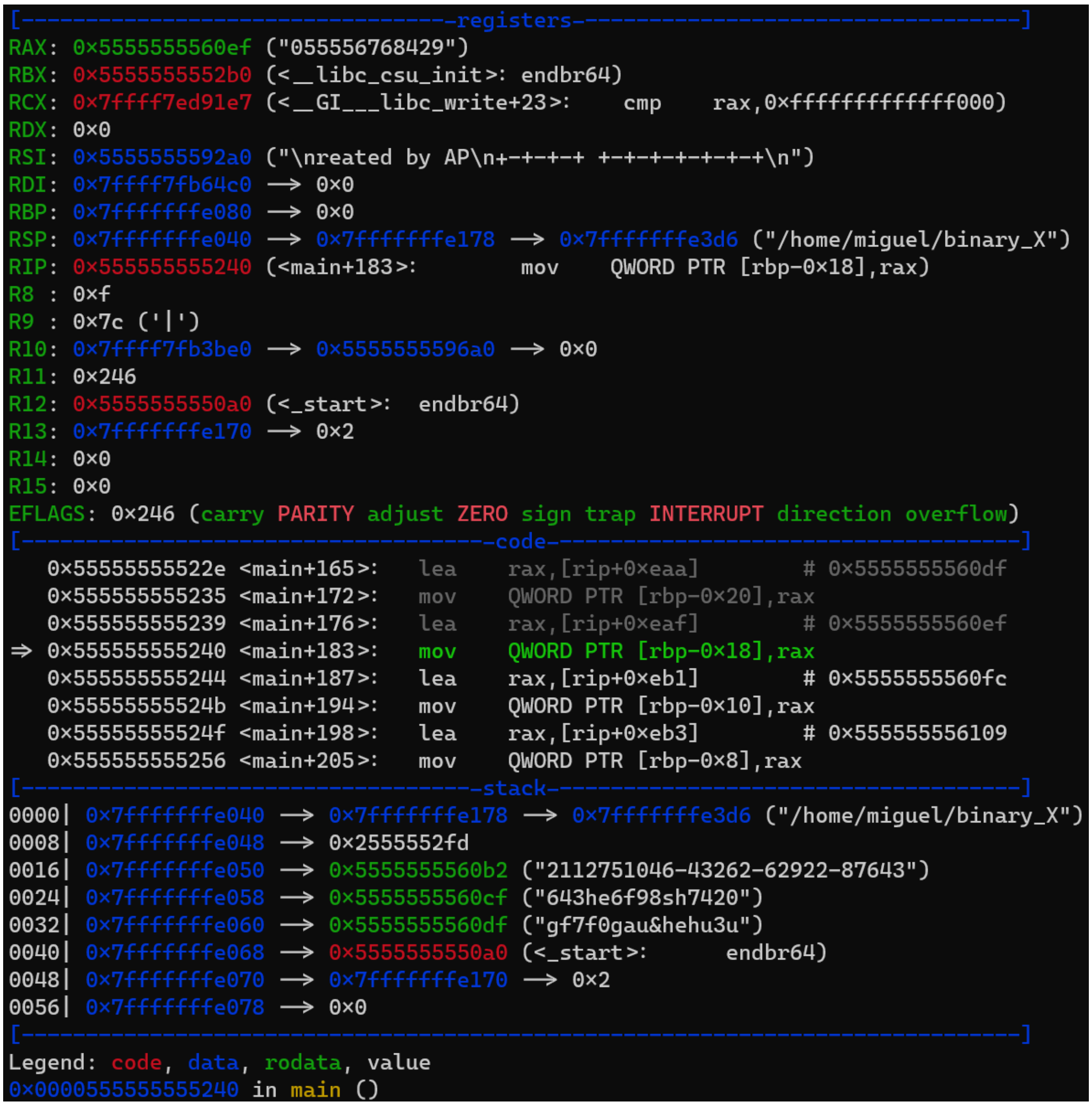
After the break point, we can run the program with an invalid key so we can that when we are stepping through the code, we can see the invalid key being compared against the correct one, figure 11.



```
gdb-peda$ r 4141414141414141414141414141
Starting program: /home/miguel/binary_X 4141414141414141414141414141
[----------------------------------registers-----------------------------------]
RAX: 0x555555555189 (<main>:    endbr64)
RBX: 0x5555555552b0 (<__libc_csu_init>: endbr64)
RCX: 0x5555555552b0 (<__libc_csu_init>: endbr64)
RDX: 0x7fffffffe190 --> 0x7fffffffe407 ("SHELL=/bin/bash")
RSI: 0x7fffffffe178 --> 0x7fffffffe3d6 ("/home/miguel/binary_X")
RDI: 0x2
RBP: 0x0
RSP: 0x7fffffffe088 --> 0x7ffff7def0b3 (<__libc_start_main+243>:        mov    edi,eax)
RIP: 0x555555555189 (<main>:    endbr64)
R8 : 0x0
R9 : 0x7ffff7fe0d50 (endbr64)
R10: 0x7ffff7ffcf68 --> 0x6ffffff0
R11: 0x202
R12: 0x5555555550a0 (<_start>:  endbr64)
R13: 0x7fffffffe170 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x555555555179 <__do_global_dtors_aux+57>:   nop    DWORD PTR [rax+0x0]
   0x555555555180 <frame_dummy>:        endbr64
   0x555555555184 <frame_dummy+4>:      jmp    0x555555555100 <register_tm_clones>
=> 0x555555555189 <main>:       endbr64
   0x55555555518d <main+4>:     push   rbp
   0x55555555518e <main+5>:     mov    rbp,rsp
   0x555555555191 <main+8>:     sub    rsp,0x40
   0x555555555195 <main+12>:    mov    DWORD PTR [rbp-0x34],edi
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe088 --> 0x7ffff7def0b3 (<__libc_start_main+243>:       mov    edi,eax)
0008| 0x7fffffffe090 --> 0x50 ('P')
0016| 0x7fffffffe098 --> 0x7fffffffe178 --> 0x7fffffffe3d6 ("/home/miguel/binary_X")
0024| 0x7fffffffe0a0 --> 0x2f7fb0618
0032| 0x7fffffffe0a8 --> 0x555555555189 (<main>:        endbr64)
0040| 0x7fffffffe0b0 --> 0x5555555552b0 (<__libc_csu_init>:     endbr64)
0048| 0x7fffffffe0b8 --> 0x856a0313d01486d3
0056| 0x7fffffffe0c0 --> 0x5555555550a0 (<_start>:      endbr64)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000555555555189 in main ()
```

Figure 11 - Running binary with invalid key

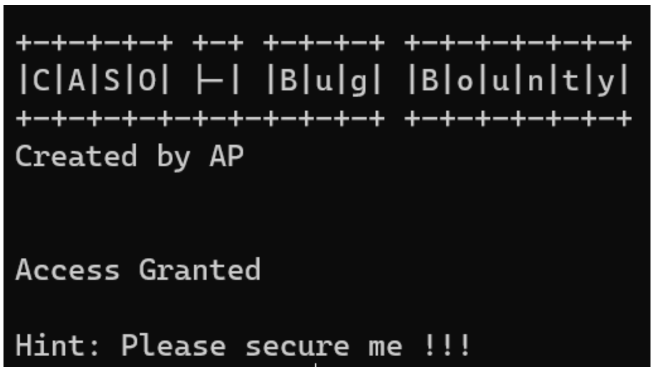
After stepping through the code several times, we can see in figure 12, in the registers section that a long number string is being stored in the RAX register which could be used as the key for the program.



```
[----------------------------------registers-----------------------------------]
RAX: 0x5555555560ef ("055556768429")
RBX: 0x5555555552b0 (<__libc_csu_init>: endbr64)
RCX: 0x7ffff7ed91e7 (<__GI___libc_write+23>:    cmp    rax,0xfffffffffffff000)
RDX: 0x0
RSI: 0x5555555592a0 ("\nreated by AP\n+-+-+-+ +-+-+-+-+-+-+\n")
RDI: 0x7ffff7fb64c0 --> 0x0
RBP: 0x7fffffffe080 --> 0x0
RSP: 0x7fffffffe040 --> 0x7fffffffe178 --> 0x7fffffffe3d6 ("/home/miguel/binary_X")
RIP: 0x555555555240 (<main+183>:        mov    QWORD PTR [rbp-0x18],rax)
R8 : 0xf
R9 : 0x7c ('|')
R10: 0x7ffff7fb3be0 --> 0x5555555596a0 --> 0x0
R11: 0x246
R12: 0x5555555550a0 (<_start>:  endbr64)
R13: 0x7fffffffe170 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x55555555522e <main+165>:   lea    rax,[rip+0xeaa]        # 0x5555555560df
   0x555555555235 <main+172>:   mov    QWORD PTR [rbp-0x20],rax
   0x555555555239 <main+176>:   lea    rax,[rip+0xeaf]        # 0x5555555560ef
=> 0x555555555240 <main+183>:   mov    QWORD PTR [rbp-0x18],rax
   0x555555555244 <main+187>:   lea    rax,[rip+0xeb1]        # 0x5555555560fc
   0x55555555524b <main+194>:   mov    QWORD PTR [rbp-0x10],rax
   0x55555555524f <main+198>:   lea    rax,[rip+0xeb3]        # 0x555555556109
   0x555555555256 <main+205>:   mov    QWORD PTR [rbp-0x8],rax
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe040 --> 0x7fffffffe178 --> 0x7fffffffe3d6 ("/home/miguel/binary_X")
0008| 0x7fffffffe048 --> 0x2555552fd
0016| 0x7fffffffe050 --> 0x5555555560b2 ("2112751046-43262-62922-87643")
0024| 0x7fffffffe058 --> 0x5555555560cf ("643he6f98sh7420")
0032| 0x7fffffffe060 --> 0x5555555560df ("gf7f0gau&hehu3u")
0040| 0x7fffffffe068 --> 0x5555555550a0 (<_start>:      endbr64)
0048| 0x7fffffffe070 --> 0x7fffffffe170 --> 0x2
0056| 0x7fffffffe078 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x0000555555555240 in main ()
```

Figure 12 - discovery of valid key

After testing program against different string variables also stored in the program as shown in figure 13, under the stack section we can conclude that the valid key is "055556768429" as when given as a parameter the resulting output is "Access Granted" shown in Figure 14.



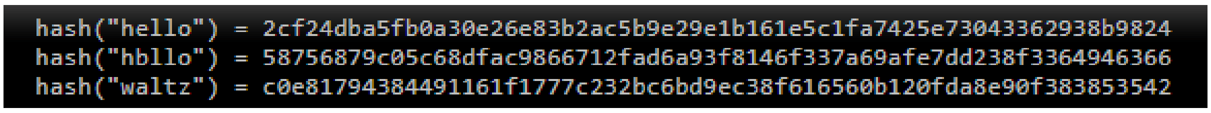
```
+-+-+-+-+ +-+ +-+-+-+ +-+-+-+-+-+-+
|C|A|S|O| |-| |B|u|g| |B|o|u|n|t|y|
+-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+
Created by AP


Access Granted

Hint: Please secure me !!!
```

Figure 13 - Output result of binary when given the correct key

ii. Propose and justify an alternative solution to mitigate this weakness in the program implementation.

When developing any program that uses sensitive information such as passwords, data should be protected in some way. In the real-world passwords are protected by employing salted password hashing, developers can implement this manually, however for most programming language there exits a module or library that would do this much more efficient and practically, this is done so anyone at any level can implement essential security features (*How to Hash Passwords: One-Way Road to Enhanced Security*, no date).

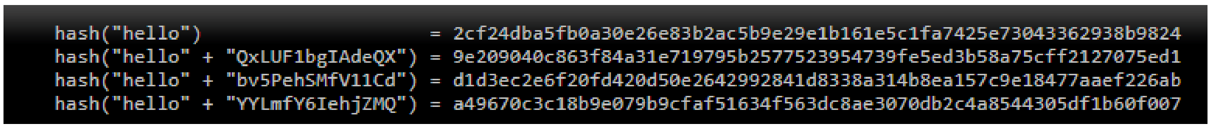
The best way to mitigate the weakness in the program above is to use a hashing algorithm along with salted values to increase security and difficulty for an attacker to crack the key. Hashing algorithms are one-way functions, they turn any data into a fixed-length "fingerprint" that CANNOT be revered, they also have the capability of detecting input changes by even one bit, resulting in a completely different hash as shown in figure 14. This is good news when dealing with passwords or keys like above, as we want a way to protect the data itself but also validate the input given by the user.



```
hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hbllo") = 58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366
hash("waltz") = c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542
```

Figure 14 - Example of a hashing algorithm

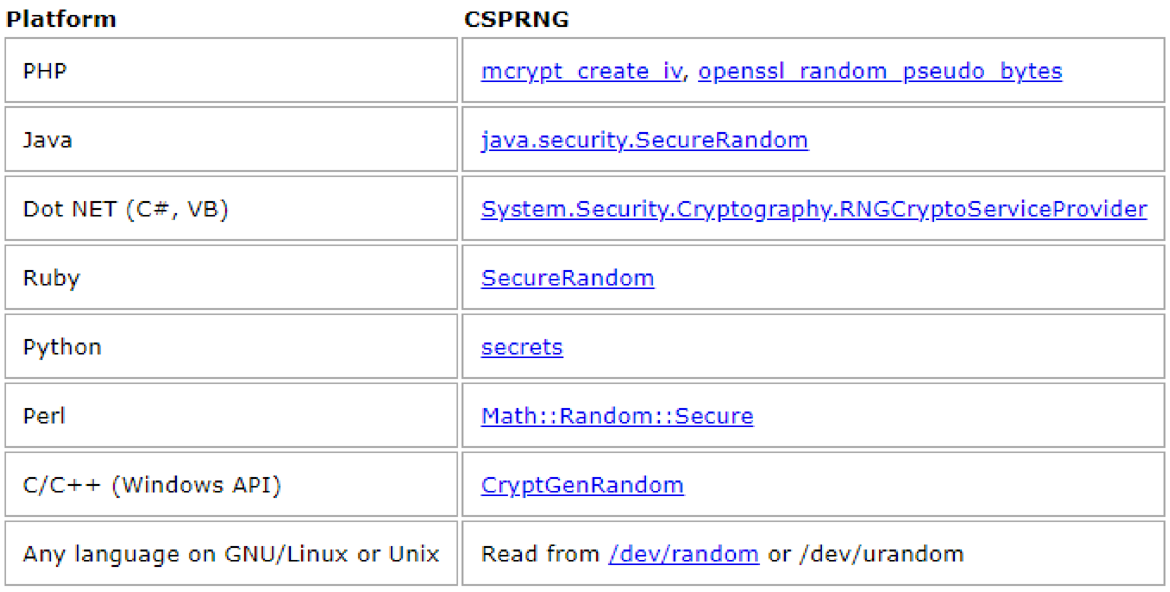
However, it is also important to note that using only hashing algorithms is not sufficient to protect against attackers, this is because attacker commonly use attacks like; rainbow table attacks, dictionary, and brute-force attacks to try and crack password hashes. For example, an attacker may create a long list of possible keys hash them and try them against the program until success. Although, it may not effective it creates a vulnerability and a chance for an attacker to gain unauthorised access, for that reason, hashes should also be implemented along a concept known as salt. The concept is simple, append a randomised string to the password before hashing. For example, you can append a random string of characters at the end of the key to make the hash random, this way attacks such as lookup tables and rainbow tables become ineffective. This is because an attacker would not be able to know in advance what the salt will be, meaning the cant pre-compute a lookup table or rainbow table (*Secure Salted Password Hashing - How to do it Properly*, no date).



```
hash("hello")                    = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1
hash("hello" + "bv5PehSMfV11Cd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab
hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007
```

Figure 15 - hashing data using salt

The correct way of implementing hashing is by using a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG). CSPRNG is designed to create a unique random salt every time it is called. For example, if the program above is implemented using the C language the "rand()" function is available which provides a high level of randomness to passwords making them unpredictable, this can be used for the key used to access the program, making it harder for an attacker to gain unauthorised access. There exists, similar CSPRNG libraries for other programming languages as shown in figure 16.



Platform	CSPRNG
PHP	mcrypt_create_iv, openssl_random_pseudo_bytes
Java	java.security.SecureRandom
Dot NET (C#, VB)	System.Security.Cryptography.RNGCryptoServiceProvider
Ruby	SecureRandom
Python	secrets
Perl	Math::Random::Secure
C/C++ (Windows API)	CryptGenRandom
Any language on GNU/Linux or Unix	Read from /dev/random or /dev/urandom

*Figure 16 - CSPRNGs available for popular programming languages (*Secure Salted Password Hashing - How to do it Properly*, no date)*

One of the ways to mitigate this problem is to use hashing for sensitive information such as the key used to grant access to the program. Meaning that instead of storing the key in "plaintext", it is much more secure to hash that key before execution. The most common hashing algorithms recommended by Google are Message Digest such as MD5 and Secure Hash Algorithm like SHA-256 (*How to Hash Passwords: One-Way Road to Enhanced Security*, no date). An implementation example of this approach is to make the program hash the key credential at run time, so even when the program is loaded into a debugger a normal user would not be able to tell which hashed string contains the desired key to access the program.

References

architecture - Security considerations of x86 vs x64 (no date) *Information Security Stack Exchange*. Available at: https://security.stackexchange.com/questions/255210/security-considerations-of-x86-vs-x64 (Accessed: 14 March 2022).

Chapter 8 - Side-Channel Attacks | Elsevier Enhanced Reader (no date). doi:10.1016/B978-0-12-812477-2.00013-7.

CVE (no date) *Common Vulnerabilitites and Exposures (CVE)*. Available at: https://cve.mitre.org/ (Accessed: 24 February 2022).

CWE - CWE-242: Use of Inherently Dangerous Function (4.6) (no date). Available at: https://cwe.mitre.org/data/definitions/242.html (Accessed: 17 February 2022).

He, Z. and Lee, R.B. (2017) 'How secure is your cache against side-channel attacks?', in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. *MICRO-50: The 50th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge Massachusetts: ACM, pp. 341–353. doi:10.1145/3123939.3124546.

How to Hash Passwords: One-Way Road to Enhanced Security (no date). Available at: https://auth0.com/blog/hashing-passwords-one-way-road-to-security/ (Accessed: 14 March 2022).

Kong, J. *et al.* (2009) 'Hardware-software integrated approaches to defend against software cache-based side channel attacks', in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 393–404. doi:10.1109/HPCA.2009.4798277.

PaX (2003a) 'Non-Executable (NX) Design and Implementation by PaX'. Available at: https://pax.grsecurity.net/docs/noexec.txt (Accessed: 21 February 2022).

PaX (2003b) 'The PaX Project'. Available at: https://pax.grsecurity.net/docs/pax.txt (Accessed: 21 February 2022).

Secure Salted Password Hashing - How to do it Properly (no date). Available at: https://crackstation.net/hashing-security.htm (Accessed: 14 March 2022).

Su, C. and Zeng, Q. (2021) 'Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures', *Security and Communication Networks*. Edited by P. Nicopolitidis, 2021, pp. 1–15. doi:10.1155/2021/5559552.

Suraj Shende (2021) *x86 vs x64 : difference between x86 and x64 Architecture*. Available at: https://www.studytonight.com/post/x86-vs-x64-what-is-the-difference-between-x86-and-x64-architecture (Accessed: 18 February 2022).

Tawil, R. (2012) 'Eliminating Insecure Uses of C Library Functions'. doi:10.3929/ETHZ-A-007215322.

technopedia (no date) *Stack Smashing*, *Techopedia.com*. Available at: http://www.techopedia.com/definition/16157/stack-smashing (Accessed: 24 February 2022).

yida (2020) 'What is x86 Architecture and its difference between x64?', *What is x86 Architecture and its difference between x64?*, 24 February. Available at: https://www.seeedstudio.com/blog/2020/02/24/what-is-x86-architecture-and-its-difference-between-x64/ (Accessed: 17 February 2022).