This document is intended to be exported as Anki flashcards.
Answers generated/derived by/from AI are coloured in blue! (Gemini/NotebookLM in use)
"Open-book" answers are marked as purple, corrections from model solutions are coloured as green.
Any arguments arising from group discussion are marked as fuchsia.

7CCSMSEN
Practice Questions

If you spot anything wrong, please let me know!
discord: @insertish
izzy · insrt.uk

Contents

1. Week 2: Stack frames and function invocation

In the context of systems security as outlined throughout the module, memory errors refer to...

1. ✗ Errors that RAM modules are affected by
2. ✓ A broad class of software (implementation) vulnerabilities
3. ✗ None of the others
4. ✗ Errors that affect software developers in remembering correct (or useful) software implementation patterns

What information should we gather to exploit a buffer overflow vulnerability?

1. ✓ The address of the code we want to execute
2. ✗ None of the others
3. ✗ The value code pointers point to
4. ✓ The location of suitable code pointers

In the context of memory error exploitation, the stack is a...

1. ✓ Memory (data) region
2. ✗ Data structure
3. ✗ None of the others
4. ✗ Code region

Under normal (i.e., non-malicious) circumstances, what does a function stack frame hold?

1. ✗ Parameters to the function, the address where to hijack the execution to once the function terminates, local variables to the function
2. ✗ Parameters of the function's caller, the address where to hijack the execution to once the function terminates, local variables to the function
3. ✗ Parameters of the function, the address where to hijack the execution to once the function terminates, global variables to the function
4. ✓ Parameters to the function, the address where to resume the execution once the function terminates, local variables to the function

What does the instruction "`mov $0x2, %eax`" do?

1. ✗ It copies the 32bit value of the memory address `0x2` into the register `%eax`
2. ✗ It moves the 32bit value of the memory address `0x2` into the register `%eax`, deleting it from the original address local variables to the function
3. ✓ It copies the constant 32bit value `0x2` into the register `%eax`
4. ✗ It moves the constant 32bit value `0x2` into the register `%eax`, deleting it from the accumulator register

Which of the following IA32 assembly instructions has NULL bytes in its encoding?

1. ✗ `mov $0x1, %al`
2. ✓ `mov 0x1, %eax`
3. ✗ `movb $0x1, (%eax)`
4. ✗ `movl 0x1, (%eax)`

What does the instruction "`lea (%eax), %ebx`" do?

1. ✗ It copies the content of `%ebx` into wherever `%eax` is pointing to
2. ✗ It copies the content of wherever `%eax` is pointing to and it stores it into the register `%ebx`
3. ✓ It takes the address of wherever `%eax` is pointing to and it stores it into the register `%ebx`
4. ✗ It copies the content of `%ebx` into the address of wherever `%eax` is pointing to

2. Week 4: Memory Corruption Vuln. (pt. 1)

What are the typical components of an injection vector in the context of remote buffer overflow vulnerability exploitation?

1. ✗ machine code, sequences of addresses. Including a NOP sled is a mistake as that would decrease the likelihood of success.
2. ✗ A NOP sled, the shellcode, and a sequence of addresses enough to overflow the buffer and overwrite a code pointer on the stack (e.g., the saved return address). These components must be in this strict order
3. ✓ A NOP sled, the shellcode, and a sequence of addresses enough to overflow the buffer and overwrite a code pointer on the stack (e.g., the saved return address). These components may be in any order but the NOP sled must precede the shellcode.
4. ✗ A NOP sled, an assembly program that represents the shellcode, and a sequence of addresses enough to overflow the buffer and overwrite a code pointer on the stack (e.g., the saved return address). These components may be in any order but the NOP sled must precede the shellcode.

A shellcode represents machine code that…

1. ✗ None of the others
2. ✗ … ultimately fetches malware from the Internet (e.g., the Torpig botnet)
3. ✓ … ultimately invoke one or more system calls attackers rely on to perform arbitrary actions
4. ✗ … ultimately invoke always a function (e.g., execl) attackers rely on to perform arbitrary actions

Is it always important to make sure we don't have NULL bytes in our shellcode? Why?

1. ✓ Yes as long as the vulnerability is caused by strcpy-like functions. These functions terminate the copy upon encountering a NULL byte in the source operand (NULL bytes terminates strings in C).
2. ✗ None of the others.
3. ✗ No, NULL bytes have nothing to do with the shellcode. They might be part of the encoding of some instructions but that's fine.
4. ✗ Yes, always. NULL bytes terminates the execution of the shellcode.

What happens to data in excess copied in a buffer?

1. ✓ It depends on where the buffer is allocated; for instance, if on the stack, overflows may cause the corruption of code pointers stored on the stack
2. ✗ It depends on where the buffer is allocated; for instance, if on the heap, overflows may cause the corruption of code pointers stored on the stack
3. ✗ It depends on where the buffer is allocated; for instance, if on the stack, overflows may cause the corruption of code pointers stored on the global data
4. ✗ Unless the buffer is local (on the stack), nothing - it's simply discarded

Consider the following C code snippet.

```c
int main(int argc, char **argv) {
  char buf[512];
  snprintf(buf, sizeof(buf), argv[1]);
  return 0;
}
```

What's wrong with this code from a security perspective?

1. ✗ The program does not output the content of buf (and buf is not zeroed out).
2. ✓ The program is vulnerable to a memory corruption vulnerability (although it does not look like a buffer overflow)
3. ✗ The program suffers from a buffer overflow vulnerability
4. ✗ The program is not vulnerable to a memory corruption vulnerability because strncpy terminates the copy after 512 bytes

Assuming executable data regions, local memory corruption exploitations may facilitate the attack success because...

1. ✗ ... attackers don't need to provide shellcode as they can execute commands directly on the system
2. ✗ ... None of the others
3. ✓ ... attackers don't need to guess the shellcode address but can place the shellcode onto the environment (knowing how the kernel sets up the stack when the process starts, would help to compute the precise address the shellcode will be)
4. ✗ ... attackers need just to fill the injection vector with the shellcode, which will be executed locally

Consider the following C code snippet.

```c
int main(int argc, char **argv) {
  char buf[512];
  strcpy(buf, argv[1]);
  return 0;
}
```

Does the program suffer from a memory corruption vulnerability (buffer overflow)? If yes, is it exploitable? In other words: is it possible to successfully corrupt the saved return address on the stack and trigger the vulnerability, which ultimately would execute attacker code?

1. ✗ No, as the strcpy takes 3 arguments, one being the length of the destination buffer.
2. ✗ There is no buffer overflow. The function strcpy copies until the destination buffer is full. If the program were written in Java, we would have a memory corruption vulnerability.
3. ✗ Yes and no because the return statement terminates the execution of the program.
4. ✓ Yes and yes because the return statement terminates the function main.

3. Week 8: Secure Software Development

✗ One good approach to making software secure is to meet the security requirements after meeting all the functional requirements.

✗ Implementation is the second phase of the secure software development process.

✓ Security requirements are security-related goals or policies.

✗ The CIA triad refers to confidentiality, integrity, and authorization.

✓ Denial of service attacks attempt to make systems unavailable to legitimate users.

✓ Auditing is the process of logging and monitoring the activities of a system.

✗ Flaws are implementation problems, whereas bugs are design problems.

4. Selected Qs from January 2017/2018 Papers

2017 Q1i

Why is it crucial that the stack grows from higher addresses to lower ones for a buffer-overflow attack to work? (Hint: Explain carefully what is stored on the stack and how it influences the control flow of the program.) **[6]**

As the stack grows from higher to lower addresses, various information such as return addresses, local variables, frame pointers, and function arguments are placed on the stack.

Relative to the current executing function, all of this information will always be further up the stack, i.e. in the higher addresses.

Buffer overflows write past the buffer into these higher addresses hence allowing us to overwrite key information such as the return address of the executing function and hence hijack the control flow of the program.

2017 Q1k

Consider buffer-overflow attacks: When filling the buffer with a payload (for example for starting a shell), what is the purpose of padding the string at the beginning with NOP-instructions? **[4]**

Sometimes, we do not have the precise address of where our shellcode will be, so including a NOP sled allows us to estimate the address to jump to.

When we land in the NOP sled, the NOP instructions will simply do nothing and move on to the next instruction. If the shellcode is directly after the NOP sled, then it will be executed.

Therefore we can jump anywhere in the NOP sled to reach the shellcode, which gives us a range of addresses we can try / increase the probability we reach the shellcode.

2017 Q1e

In the context of security validation, what is the main difference between hacking and penetration testing? **[2]**

Hacking is the exploitation of vulnerabilities in a system by a third party threat actor who does not have permission to do so, meanwhile penetration testing is the intentional exploitation/finding of flaws in a system by the first party to try to patch up security issues.

2017 Q1f

(1) What are the differences between a penetration test and a vulner- ability scan? Give 2 facts and explain your answers. (2) What are the similarities between a penetration test and a vulnerability scan? Give 2 facts and explain your answers. **[8]**

They are different in:

- Scope/Depth: Vulnerability scanning uses automated tools to find potential vulnerabilities or flows, while penetration testing involves exploitation of identified vulnerabilities to gain further access and assess potential impact.
- Methodology: Vulnerability scanning tries to target any potential vulnerability automatically while penetration testing is done manually and assess individual flaws.

They are the same in:

- Goal: Both penetration testing and vulnerability scanning involves finding flaws with a system in order to then make it more secure.
- Usage: They are both part of a typical security assessment, and often used in conjunction.

2017 Q1j

A Unix directory is defined as follows:

```shell-unix-generic
$ ls -ld . * */*
drwxr-xr-x 1 alex staff 32768 Apr 2 2010 .
-rw----r-- 1 alex gurus 31359 Jul 24 2011 manual.txt
-r--rw--w- 1 marc gurus 4359 Jul 24 2011 report.txt
-rwsr--r-x 1 marc gurus 141359 Jun 1 2013 microedit
dr--r-xr-x 1 marc staff 32768 Jul 23 2011 src
-rw-r--r-- 1 marc staff 81359 Feb 28 2012 src/code.c
-r--rw---- 1 lisa gurus 959 Jan 23 2012 src/code.h
```

With group memberships:

- staff: alex, marc, lisa
- gurus: lisa

The file microedit is a text editor, which allows its users to open, edit and save files. Note carefully that microedit has set its setuid flag, indicated by the lower-case s. Fill in the access control matrix below that shows for each of the above five files, whether alex, marc, or lisa are able to obtain the right to read (R) or replace (W) its contents using the editor microedit. **[15]**

	euid	egid(s)	**manual.txt**	**report.txt**	**microedit**	**src/code.c**	**src/code.h**
alex	marc	staff	R (other)	R (user)	R,W (user)	R,W (user)	- (other)
marc	marc	staff	R (other)	R (user)	R,W (user)	R,W (user)	- (other)
lisa	marc	staff, gurus	- (group)	R (user)	R,W (user)	R,W (user)	R,W (group)

Proof of (user, group-but-not-user, other) permission precedence by use of Linux machine:

```sh
izzy@redstone:~$ touch testing
izzy@redstone:~$ chmod 000 testing
izzy@redstone:~$ cat testing
cat: testing: Permission denied
izzy@redstone:~$ echo test > testing
-bash: testing: Permission denied
izzy@redstone:~$ chmod u+x testing
izzy@redstone:~$ echo test > testing
-bash: testing: Permission denied
izzy@redstone:~$ cat testing
cat: testing: Permission denied
izzy@redstone:~$ chmod g+r testing
izzy@redstone:~$ cat testing
cat: testing: Permission denied
izzy@redstone:~$ stat testing
Access: (0140/---xr-----)  Uid: ( 1000/    izzy)   Gid: ( 1000/    izzy)
```

Also note, `chmod` allows use of `a+r`/`a+w` to give everyone the permission.

2018 Q1j

Why does randomising the addresses from where programs are run help with defending against buffer-overflow attacks? **[2]**

Address randomisation prevents attackers from deterministically finding certain addresses such as where the environment, arguments, or stack is. As such, this increases the difficulty of finding viable addresses to jump to after a successful buffer-overflow.

5. January 2019 Paper

Consider the following code fragment

```c
int main(int argc, char ** argv) {
  (void) foo(argv[1]);
  exit(0);
}

int foo(char * arg) {
  return bar(arg);
}

int bar(char * arg) {
  char lbuf[1024];
  if (strlen(arg) >= 1024)
    return -1;

  memset(lbuf, 0, sizeof(lbuf));
  puts("Welcome: ");
  sprintf(lbuf, arg);
  write(STDOUT_FILENO, lbuf, strlen(lbuf));
  fflush(stdout);

  return 0;
}
```

Q1a

Give a thorough description of the program's vulnerability. In particular, name the vulnerability (1 mark) and provide a detailed overview of its exploitation (3 marks). Then, identify and explain thoroughly all the components that are involved in the exploit (4 marks)

- Vulnerable to string formatting attack
- An attacker would:
  - Specify a specially crafted string as the program's first argument (hence `argv[1]`)
  - This argument is passed through to foo, where it is passed to bar, where it is passed to `sprintf`.
  - As it is the second argument to `sprintf`, it will be used as the format string.
  - The specially crafted string may contain any format specifier the attacker wishes to use (whether to read or write memory).
  - An attacker can overwrite the return address on the stack using format string specifiers to redirect program execution to malicious shellcode.
  - The attacker would also need to place the shellcode in memory, potentially using the `arg`.
- The components involved are:
  - `argv`: the program's arguments are user input and controlled by attacker (injection vector)
  - `sprintf`: a string format function that accepts a buffer to write to and a format string the attacker controls the format string used here (this is the vulnerable component)
  - format str specifiers: malicious components inserted by attacker into string
  - stack: stack is used to store function call data, return address lives here (our target)
  - shellcode: malicious code injected into program's memory that attacker wants executed
  - return address: address on the stack overwritten to redirect execution flow

Q1b
How would an attacker exploit the vulnerability (*string formatting vulnerability*)? Hint: describe in detail what the injection vector would look like (and what retaddr and retloc the attacker may use). Use symbolic values and addresses when needed (no need to write down the shellcode). **[10]**

The attacker may take the strategy of overwriting a return address or other important code pointer relating to the execution flow.

The injection vector must at least include (or have encoded in it):
- The address of where we would like to jump to (retloc)
- The code pointer we'd like to overwrite (retaddr)
- Any format specifiers required to achieve arbitrary write of the aforementioned data

The injection vector may include the shellcode, which itself may be preceded by a NOP sled to make it easier to guess the address that the shellcode lands in.

The attacker can use Write4 primitives to produce an injection vector that is composed of:
- A list of four addresses of retaddr, retaddr+1, retaddr+2, retaddr+3
  These will be used later to write over retaddr, we can find their offsets by using format specifiers that read data from the stack at specified offsets and comparing if they match.
- A repeated series of padding (writing characters) and writing (of 'characters written so far').
  There is a format specifier that can write the characters written so far (truncated to one byte) to a given address at an offset in memory.
  We use the offsets found earlier and we can use a format specifier to write an arbitrary number of characters to control the 'characters written so far' value to then write over any value we want in memory, one byte at a time.

The attacker may pick a number of different values for retaddr:
- Could use the return address from the current function which is further up the stack.
- Could find the GOT entry for the `write` library call and hijack the call.

The attacker may pick a number of different values for the retloc, such as a location on the stack (if the shellcode is included in the injection vector) or a location in `envp` or `argv` if they control the environment/execution of the program.

Q1c
Context: string formatting exploit using `sprintf`
Would StackGuard or a bounds checker mitigate the vulnerability (1 mark)? Explain clearly the reasons (4 marks)

No.

StackGuard works by placing canaries on the stack and then making sure they don't change before the function returns. The proposed attacks precisely modify the memory required so we wouldn't happen to change a canary, triggering program termination.

In the case of a bounds checker, our injection vector will not be longer than the buffer we are writing to anyways, so we wouldn't have any reason to trigger it.

Neither would mitigate this vulnerability, as the vulnerability exploits the format string of the `sprintf` function itself while the function itself is behaving as expected by both StackGuard and the bounds checker (writing to the extent of the buffer and not beyond). Both of these defenses do not protect against format string attacks which do not write past the end of a buffer, but instead write to arbitrary memory addresses specified by the attacker.

NB. **actually** with a small enough buffer, you could hit bounds or overflow! %192u, etc write garbo!!

Q1d
Context: vulernable line of code `sprintf(lbuf, arg);`
How can the program be fixed?

The programmer could either:
- Use a safe string copy function.
- Specify a safe format string such as `"%s"` and then provide `arg` as the 3rd argument: `sprintf(lbuf, "%s", arg);`
- Sanitise the user input however this could be complex to implement

Q2ai *Access Control*
What is a subject in access control terminology? Briefly justify your answer or provide an example. **[1]**

A subject is an active entity that wishes to access resources, e.g. a running process, a user.

Q2aii *Access Control*
What is the decision to grant or deny an access to a process based on? Briefly justify your answer or provide an example. **[1]**

Decision to grant or deny access to a process is based on the rights and permissions associated with the subject making the request and the security policy in place. e.g. the reference monitor checks whether the principal bound to the subject has right to access the object.

Q2aiii *Access Control*
What is an Access Control List? Briefly justify your answer or provide an example. **[1]**

A list of principals permitted to have access to an object with their respective permissions which is attached to a subject. For example, an object may have a list [(Dave, R+W)] which allows only the principal Dave to read and write the object.

Q2aiv *Access Control*
What is associated a capability (i.e., a list of objects) to? Briefly justify your answer or provide an example. **[1]**

It is associated with a subject.
For example, the user (subject) could have the list [(A, R+W), (B, R)] which means they can read and write A, read B, and cannot access anything else.

Q2av *Access Control*
Describe the main properties of discretionary and mandatory access control policies. **[4]**

Discretionary Access Control:
- Ownership of resources is used to determine access rights
- Resource owners have discretion to grant/revoke access to their own resources
- Individual users manage access to their resources

Mandatory Access Control:
- Access to objects is determined by security domain of subjects (access is only allowed if the object and subject belong to the same security domain)
- A central authority defines the security policy and enforces it on all subjects.

Q2bi

Suppose that in a system that enforces an information flow policy for integrity, subjects s1 and s2 have high security clearance (inherited from the associated principal), while objects o1, o2, o3, and o4 all have medium security sensitivity. Subjects s3 and s4 have low security clearance.

Using a diagram, indicate the allowed direction of information flow for this example, and show all the possible operations by all subjects/objects of the example by indicating whether each operation is allowed or denied by the example policy. **[9]**

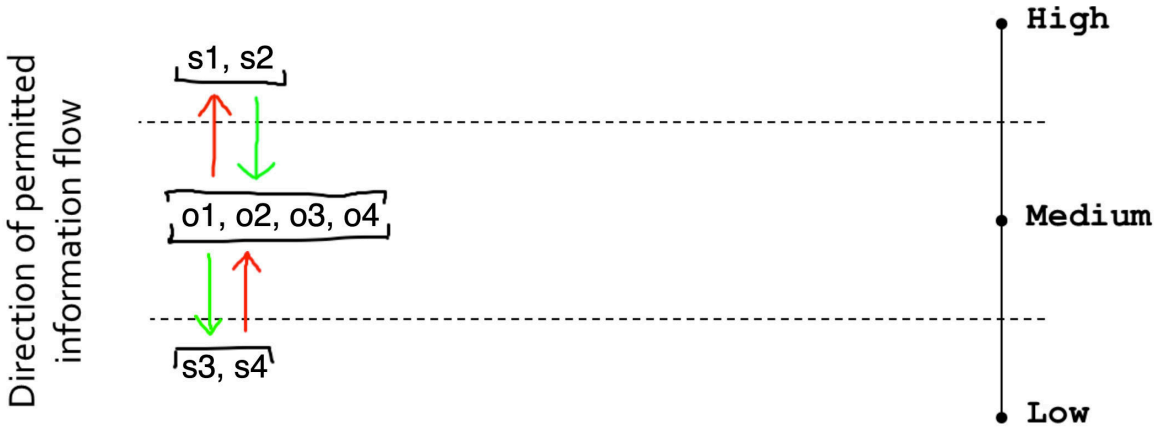


Figure 1: Information Flow Policy

Subjects s_1 and s_2 can write to any of the objects o_1, o_2, o_3, o_4.
Subjects s_1 and s_2 can are not permitted to read from any of the objects o_1, o_2, o_3, o_4.
Subjects s_3 and s_4 are not permitted to write to any of the objects o_1, o_2, o_3, o_4.
Subjects s_3 and s_4 can read from any of the objects o_1, o_2, o_3, o_4.

Q2bii

How are the read and write operations generically defined in terms of information flow?

- A read operation causes information to flow from the object to the subject.
- A write operation causes information to flow from the subject to the object.

Q2ci
"In Unix-like operating systems, chmod is the command and system call which may change the access permissions to file system objects (files and directories). It may also alter special mode flags."

Briefly describe the classes of operations/permissions and users in Unix-like OSes used for file access control. **[3]**

A user may perform a read, write, or execute operation on a file.
When determining a user's access to a file, we check whether they are:
- the owner (u): hence (exclusively) use the permissions specified for the owner
- part of the group (g): hence (exclusively) use the permissions specified for the group
- anyone else (o): hence (exclusively) use the permissions specified for everyone else

Q2cii
Describe an example of a special mode flag in Unix-like OSes **[3]**

An example special mode flag is the setuid/setgid flags, which upon execution of the file, will set the user ID or group ID during execution so that the file (program) will act as if it is that specified user or group instead of the actual user or group.

For example, a program owned by insert and group staff with the flag setuid is run by a user user-b in group-b. The program will run as if it were being executed by insert in group-b.

Another special mode flag is the sticky bit:
- When set on a file changes swapping behaviour.
- When set on a directory, it will restrict deletion/rename of the directory to the owner.

Q3a
Consider the following C code fragment:

```c
int main(int argc, char ** argv) {
  char foobar[1024];
  if (argc > 1)
  strncpy(foobar, argv[1], strlen(argv[1]));
  exit(1);
}
```

Does the program suffer from a memory corruption vulnerability? If not, explain the reasons. If yes, is it possible to successfully exploit this vulnerability? In other words, is it possible to provide specific input to such a program to take advantage of its vulnerability and thus execute arbitrary code (for instance, spawning a shell), on x86-32 architectures? If yes, explain how you would exploit it (high-level steps). If not, explain why and what you would change in the code to make it exploitable. **[5]**

Yes, it is vulnerable to a buffer overflow memory corruption.
Even though `strncpy` has a length check, it uses the length of the arbitrary user input `argv[1]`, which means the data from `argv[1]` will be copied into `foobar` and will overwrite the stack after 1024 characters have been written.

However, we cannot exploit it in this case.
We can overwrite the return address which is present in the stack just after the buffer, but we will never return from this function (as it calls exit syscall instead) so we will never be able to achieve an arbitrary jump.

To make it exploitable, remove the `exit(1)` at the end of the function to allow us to jump to retaddr.

Q3bi

Assuming that the code is vulnerable (buffer overflow to overwrite return address) and that the vulnerability can be successfully exploited, then consider the following x86 assembly code fragment, which may be used to exploit the previous vulnerability:

```asm
jmp ahead
back:
  popl %ebx
  movl %ebx, 0x8(%ebx)
  movl $0x0, %eax
  movb %al, 0x7(%ebx)
  movl %eax, 0xc(%ebx)
  movl %eax, %edx
  movl $0xb, %eax
  leal 0x8(%ebx), %ecx
  int $0x80
ahead:
  call back
  .string "/bin/sh"
```

Assuming the above assembly snippet will be placed on the stack, what does the assembly code do? Add comments to each line and draw the stack layout **before** and **after** the considered instruction is executed.

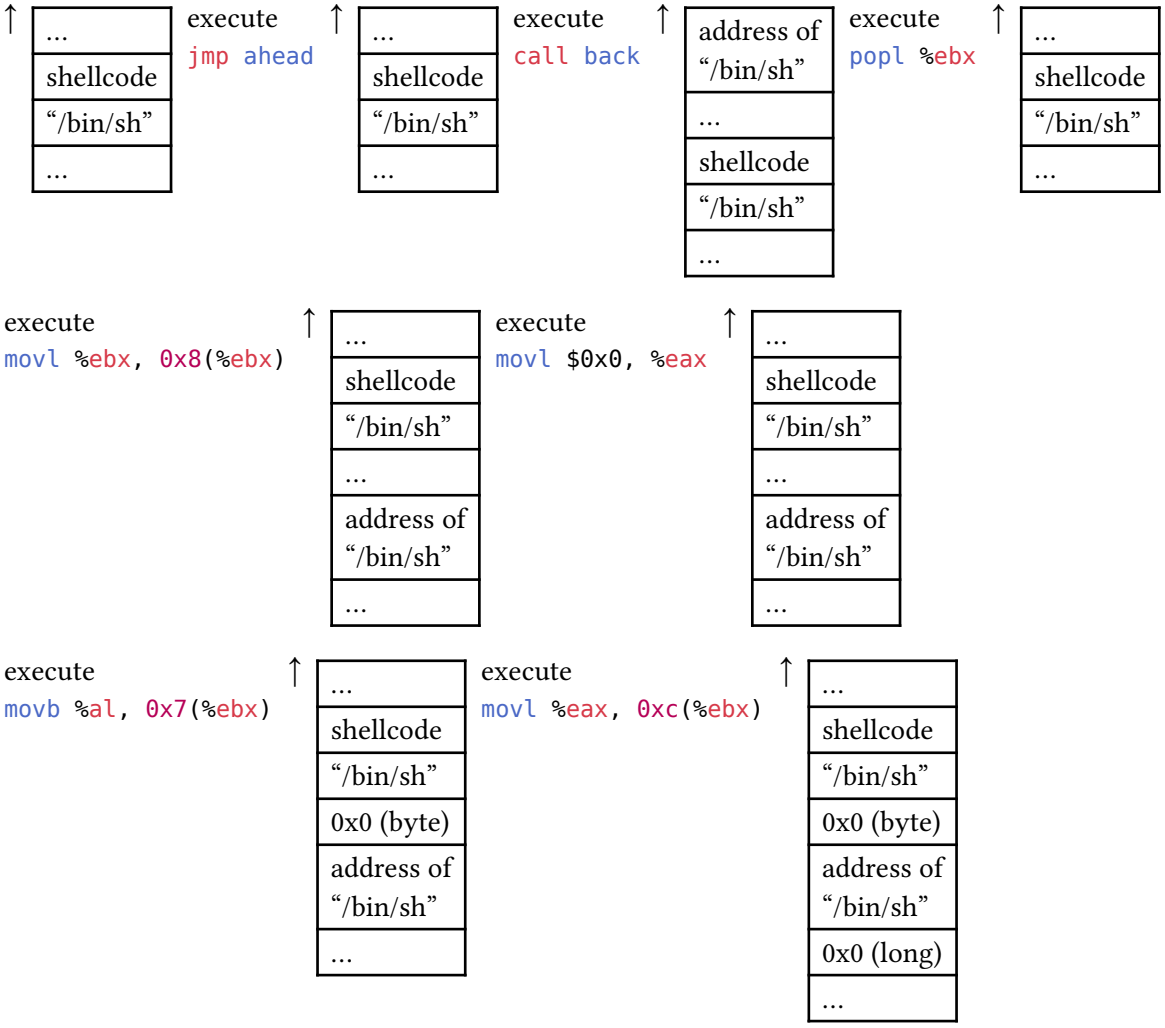
Note: you should clearly point out the direction the stack is growing towards. **[12]**

```asm
jmp ahead ; jump to the call instruction at ahead label
back:
  popl %ebx ; pop the address of the string into %ebx (1st arg. to execve)
  movl %ebx, 0x8(%ebx) ; copy the address to offset 8 from the string in memory
  movl $0x0, %eax ; write NULL value to %eax
  movb %al, 0x7(%ebx) ; copy byte of this NULL value to offset 7 (from str...)
  movl %eax, 0xc(%ebx) ; copy long of this NULL value to offset 12 (...)
  movl %eax, %edx ; place NULL long value in %edx (3rd arg. to execve)
  movl $0xb, %eax ; place 0xB syscall index in %eax for int instruction below
  leal 0x8(%ebx), %ecx ; copy address of address of string to %ecx (2nd arg.)
  int $0x80 ; execute execve using the three provided arguments
ahead:
  call back ; jump to second instruction (the back label)
            ; and push the address of the string to stack
  .string "/bin/sh"
```

Diagrams included below showing changes on stack.

For the purposes of the following diagrams:

- Since we don't know where the shellcode is 'placed' on the stack, "..." indicates zero or more memory locations between adjacent elements in the stack.
- The top of the stack is the first element, and it grows upwards (towards the top).



The remainder of instructions do not affect the stack.

Q3bii

An attacker creates a suitable injection vector to exploit the aforementioned memory error (**Q3**). To this end, he places the shellcode (**Q3b**) in the injection vector, pads it with his initials so as to create a message long enough to overflow lbuf; then the attacker adds the appropriate address at the right place and terminates the message with a NULL ('0'). In other words, the injection vector looks as follows:

```
+-----------------+-------- ---------+-----------+----+
|  shellcode  | lclclc ... lclclclc | 0xbff00b4r | \0 |
+-----------------+-------- ---------+-----------+----+
```

Next, he runs the program giving this injection vector to it as its first argument. To his surprise, the attack fails. He asks you for help. State why the attack cannot work. **[2]**

The shellcode provided in **Q3b** includes one or more NULL bytes. The string copy function used will stop copying characters when it reaches a NULL byte! One such culprit is `movl $0x0, %eax` whose instruction will most certainly include four NULL bytes for the first operand.

Q3biii
Show how the shellcode can be modified to make the attack pos- sible. (Note: don't worry if you do not know the exact syntax of instructions; marks will be awarded for a clear explanation.) **[2]**

Culprit: `movl $0x0, %eax`

They need to eliminate all NULL bytes from the payload, they can achieve this by for example, constructing NULL bytes or longs using XOR operations, e.g. they could xor the value of %eax with %eax, then use the value now stored in %eax as opposed to using immediate values.

Q3c
Assume the aforementioned code is vulnerable (or can be modified to be so) and the vulnerability can be successfully exploited. State and describe what technique(s) would an attacker use to exploit the vulnerability shown at the beginning of the question, if the kernel enforces a non-executable stack protection (again, assume the small program shown at the beginning of the question is exploitable or can be modified to be exploited successfully)? **[4]**

There are two strategies we can employ here:

- return-to-libc attack: we overwrite the stack frame to prepare the return address to point to a function loaded from libc, such as `memcpy` or `system` which, followed by the arguments we'd like it to use. These functions are not on the stack so we won't be prevented from executing them.
- Return-Oriented Programming: a more generalised form of return-to-libc involves gathering 'gadgets' from the program and loaded libraries that can serve a specific purpose, such as an instruction to set a certain register or some other action which is always followed by a `ret` instruction that allows us to chain arbitrary actions together (assuming we can find the gadget for them) to then build more complex behaviour. This works because whenever ret is executed the stack pointer moves back as the return address is consumed, so it can simply move to using our next return address in sequence.

Q4ai/ii/iii

Partly covered, i.e. command injection, but pretty simple to answer:

1. How does XSS work? **[4]** Untrusted user input is used in such a manner where they their input is injected into the page without any validation or sanitisation, hence allowing arbitrary execution of code on the page. Whether for the current user or another user visiting the site, viewing the attacker's payload in some context.
2. State what information an attacker can steal using XSS and why is it useful. **[3]** Depending on the context, the attacker could scrape login credentials or other sensitive information by monitoring what the user does on the page, or in fact calling to other parts of the website and fetching information that it needs by itself.
3. How can the effects of XSS be mitigated? Please outline limitations as well, if any. **[3]** The go-to solution for most developers is sanitisation, which is built into most web frameworks these day, however these must be very comprehensive otherwise attackers may find workarounds. Alternative half-solutions include Content Security Policy, which can be used to stop inline scripts entirely (prevention), or block calls to external websites (stop data exfiltration).

Q4bi/ii/iii/iv

Partly covered, i.e. command injection, but pretty simple to answer:

1. Briefly describe how SQL Injection works. **[4]** When constructing SQL statements, developers fail to sanitise user inputs when inserting them into queries, hence allowing attackers to execute arbitrary SQL commands and sometimes see their output directly.
2. Apart from username and password input fields, which variables are candidates for SQL Injection? **[3]** Any plaintext value that is interpolated into an SQL query without santisation is a candidate for an SQL injection.
3. What techniques can an application programmer use to mitigate the effects of SQL injection attacks? Please outline limitations as well, if any. **[4]** Sanitisation, generally 100% effective, the main concern is quotation marks.
4. skipped, 100% not exam material

6. January 2021 Paper

Q1a

List, and explain, two benefits of static analysis. **[2]**

- Static analysis can provide a much higher coverage than traditional testing, the program's different execution paths are automatically inferred rather than having someone manually go through and make sure all cases are covered.
- Static analysis can be performed without running the program, allowing analysis to be done early in software development process, potentially even on incomplete programs.

Q1b

List, and explain, three drawbacks of static analysis. **[3]**

- Static analysis can be time-consuming to run, and often quite computationally difficult as it has to reason about all of the execution paths.
- We can't achieve perfect static analysis because of the Halting problem, so compromise in the sense that it may miss some errors or have false-positives since it cannot perfectly reason about the program provided. [AI generated answer split these into two.]
- We can only reason about a limited number of properties, so we can only capture so many types of errors present in the program.

Q1c

What impact does static analysis have on software development process? **[4]**

It would not have any impact on requirements analysis or design, however it would impact:

- Implementation: developers can use static analysis while building their software to continuously check that the components they are implementing are as error-free as possible
- Testing: static analysis can also be used on the final program after it is fully implemented to ensure all components function as expected
- Static analysis can be used early in the software development process, as it does not require the program to be executed.
- It can lead to higher quality software and be used as a tool to support code review.

Q1d

What does it mean for static analysis to be sound? **[2]**

This means that if the static analysis claims a program is error-free, then it truly is error-free.

Q1e

What does it mean for static analysis to be complete? **[2]**

This means that if the static analysis claims a program has one or more errors, it does truly have those errors present.

Q1g

```c
int printf(untainted char * fmt, ...);
void read(int, tainted char * input, int);

char name[10];
read(0, name, sizeof(name));
char *x;
x = name;
x = "hello!";
printf(x);
```

We are interested in analysis that identifies no tainted data flows (where untainted < tainted in a lattice). Given the initial taint source and untainted sink:

1. Show how the program would be changed if we carried out a flow-sensitive static analysis, assuming flow-sensitive and path-/context- insensitive analysis. **[1]**

   The analysis would track the different assignments to `x`: so on line 7 `x` would be tainted, but on line 8, `x` is no longer tainted. Meanwhile with flow-insensitive analysis, `x` would have a single taint status and always be marked as untainted. The program isn't literally modified but flow-sensitive analysis tracks changes to the taint status!

2. Create a name for each missing type qualifier, assuming a flow-sensitive analysis, assuming a flow-sensitive and path-/context-insensitive analysis. **[2]**

   Let's mark the following:
   - All instances of `name` as δ
   - First assignment of `x` as α
   - Second assignment of `x` as β
   - `fmt` is simply marked untainted
   - `input` is simply marked tainted

3. For each statement in the program, generate constraints on possible solutions. **[1]**

   - $\text{tainted} \leq \delta$
   - $\delta \leq \alpha$
   - $\text{untainted} \leq \beta$
   - $\beta \leq \text{untainted}$

4. Solve the constraints to produce solutions for the type qualifiers identified earlier **[1]** and state whether the resulting flow is legal or illegal **[1]**.

   There is a possible solution, hence the flow is legal:
   - $\delta, \alpha = \text{tainted}$
   - $\beta = \text{untainted}$

Q2a
Name the vulnerability **[1]**, provide a detailed overview of its exploitation **[3]**. Then identify and explain all components involved in the exploit. **[4]**

```c
int main(int argc, char ** argv) {
  int n;
  if (argc == 2) n = foobar(argv[1]);
  else exit(1);
  printf("Hey! %s, nice to meet ya (%d)\n", argv[1], n);
  exit(n);
}

int foobar(char * arg) {
  char msg[512];
  memset(msg, 0, sizeof(msg));
  strcpy(msg, "Hey! ");
  snprintf(msg + strlen("Hey! "), sizeof(msg) - strlen("Hey! "), arg);
  return strlen(msg);
}
```

The program has a string formatting vulnerability.
An attacker could specify a specially crafted string as the first argument to the program `argv[1]`, which is then passed to the `foobar` function in which it is used as the format string for `snprintf`. Since the attacker controls the format string, they can arbitrarily read memory at any given offset in the stack and arbitrarily write bytes anywhere into memory assuming the buffer we are writing into is accessible on the stack (to use Write4 primitives). So they could overwrite the return address on the stack and jump to any arbitrary code that they could also include in their message or otherwise inject through the environment (if this is a local attack).

Components involved are:
- `argv`/`arg` is an untrusted string controlled by the attacker
- `snprintf` is the function we call that accepts format strings that is the vector for abuse
- Format string specifiers allow us to perform various operations such as read integers or strings from the arguments of `snprintf` (as the stack grows down, unspecified arguments simply point up the stack), write any arbitrary number of characters, or write the number of characters written so far to an address provided by a value at a given offset on the stack.
- The return address in the stack frame of `foobar` is a memory location we can overwrite in order to control the program's execution flow.

Q2b
How would an attacker exploit the vulnerability? Hint: describe in detail what the injection vector would look like (and what retaddr and retloc the attacker may use). Use symbolic values and addresses when needed (no need to write down the shellcode). **[10]**

Already described this in the 2019 paper, see Q1b.

Q2c
Would bounds checkers mitigate the vulnerability **[1]**? Explain clearly the reasons **[2]**

Already described this in the 2019 paper, see Q1c.

Q2d
Would StackGuard mitigate the vulnerability **[1]**? Explain clearly the reasons **[2]**

Already described this in the 2019 paper, see Q1c.

Q2e
How can the program be fixed? **[2]**

Already described this in the 2019 paper, see Q1d.

Q3a
Does the program suffer from a memory corruption vulnerability? If not, explain the reasons. If yes, explain the reasons and and how it is possible to successfully exploit this vulnerability. In other words, is it possible to provide specific input to such a program to take advantage of its vulnerability and thus execute arbitrary code (for instance, spawning a shell), on x86-32 architectures? If yes, explain how you would exploit it (high-level steps, including what input and size you should provide). If not, explain why and what you would change in the code to make it exploitable. **[5]**

```
1   int main(int argc, chat ** argv) {
2     if (argv[1]) return foo(argv[1]);
3     else return foo(argv[0]);
4   }
5
6   int foo(char * arg) {
7     char bar[128];
8     if (sizeof(arg) > 128) {
9       strcpy(bar, arg);
10      exit(0);
11    }
12    strcpy(bar, arg);
13    return strlen(bar);
14  }
```

Yes, the use of `strcpy(bar, arg)` can lead to a buffer oveflow if the argument is longer than the buffer `bar` (128 characters). Furthermore, the check to prevent the exploitation of this (i.e. smashing the stack to overwrite return address of `foo`) will never fire because of a logic error, rather than check if the length of the array is greater than 128, it instead checks the size of the pointer to the array, which on a 32-bit system will be at most 4 bytes, hence the check always returns false.

An attack can craft a special input to the program which consists of:
- The shellcode in the argument (or environment).
- Padding of $128 - m$ characters to fill the buffer, where m is the length of the shellcode if it has been included in the argument (hence would be written to the buffer).
- A repeating pattern of the address we would like to jump to, i.e. where our shellcode is. An attacker could deterministically find this address assuming a legacy virtual address space without address space layout randomisation, whether they place it in the environment or the buffer itself. The attacker could include a NOP sled before their shellcode to increase the range of addresses that they could jump to, in the case that they are stochastically picking the address.

Q3bi
Similar enough to Q3bi that I'm skipping it here, it's the same description of every line and then subsequent diagram drawing. An annotated copy is in the next question though!

Q3bii/iii
What is the issue with the following shellcode? **[2]** How can it be fixed? **[2]**

```asm
jmp ahead
back:
  popl %ebx
  movl %ebx, 0x8(%ebx)
  xorl %eax, %eax
  movb %al, 0x7(%ebx)
  movl %eax, 0xc(%ebx)
  movl %eax, %edx
  movl $0xb, %eax
  movl 0xc(%ebx), %ecx
  int $0x80
ahead:
  call back
  .string "/bin/sh"
```

1. Currently this shellcode sets up the registers as such before calling execve:
   - %eax = 0xb (syscall index of execve)
   - %ebx = address of '/bin/sh' — correct! this is the path parameter
   - %ecx = NULL long — wrong! this should be argc, i.e. address of address of [&'/bin/sh', NULL]
   - %edx = NULL long — wrong! this should be envp, i.e. address of [NULL]
2. So to fix the code, we must instead load the effective address of where the array of [&'/bin/sh', NULL] is present, and likewise for [NULL]!

```asm
jmp ahead
back:
  popl %ebx              ; %ebx = addr of '/bin/sh'
  movl %ebx, 0x8(%ebx)   ; place addr of '/bin/sh' offset by 2 bytes from str end
  xorl %eax, %eax        ; create 0 in %eax
  movb %al, 0x7(%ebx)    ; put NULL byte after '/bin/sh'
  movl %eax, 0xc(%ebx)   ; put NULL long after addr of '/bin/sh'
  ; movl %eax, %edx      ; %edx = 0       (wrong!)
  movl $0xb, %eax        ; %eax = 0xb
  ; movl 0xc(%ebx), %ecx ; %ecx = &[NULL] (wrong!)
  leal 0x8(%ebx), %ecx   ; corrected code: %ecx = &['/bin/sh', NULL]
  leal 0xc(%ebx), %edx   ; corrected code: %edx = &[NULL]
  int $0x80
ahead:
  call back
  .string "/bin/sh"
```

Q3c
Answered in 2019 paper, Q3c.

Q4a
A secure software development lifecycle requires security engineering to fit into all the phases of the software development process.

In which way does security engineering fit into requirements, design, implementation, and testing/ assurance? Motivate your answer. **[7]**

Security engineering is involved in every stage of secure software development:
1. Requirements: We plan security requirements, e.g. what we expect for confidentiality, integrity, availability, and ask questions about who should bew able to have access. We model potential abuse cases that could negatively affect users or the organisation.
   We create a threat model, which is the baseline capabilities of potential attackers, with which we perform architectural risk analysis, which is assessing potential attacks that can arise from potential attackers' capabilities. This can further guide security requirements.
2. Design: We continue to do architectural risk analysis while designing. We also want to incorporate security into the design of our software.
3. Implementation: We incorporate security into the implementation of the software, by using appropriate security features while programming. During development, we regularly do code review to assess bugs, errors, vulnerabilities, find memory errors, and related security flaws.
4. Testing/Assurance:

Q4b
What is a threat model and why is it important? **[4]**

The threat model is a baseline assumption about the capabilities of the attacker, which is then used to find the most likely attacks that may be performed against your assets.

~~A threat model typically details the assets we seek to protect, potential threats to assets, and risks we are willing to take or not take in regards to those assets (in other words, how much are we willing to invest to protect those assets).~~

It is important to have a well designed threat model to better aid searching of and mitigation of flaws that related to assets which are more valued.

Q4c
What is Leslie Lamport's Gold Standard? **[2]** And explain each. **[12]**

It defines mechanisms provided by a system to enforce various security requirements:
- **Authentication**: verify someone is who they claim to be and determine who is the subject of security policies
- **Authorization**: granting access rights in relation to a verified identity
- **Audit**: log and monitor activites of a system; retaining enough information to determine circumstances of breaches/misbehaviour

7. Code Sample Analysis for August 2021

2021 Q1

What is the vulnerability?

```c
void foobar(char * arg) {
  char msg[256];
  memset(msg, 0, sizeof(msg));
  if (strlen(arg) < sizeof(arg)) strcpy(msg, arg);
  snprintf(msg, sizeof(msg) - 1, arg);
  return;
}

int main(int argc, char ** argv) {
  if (argc != 2) exit(1);
  foobar(argv[1]);
  printf("Hey! Nice to meet ya %s\n", argv[1]);
  exit(0);
}
```

The vulnerability lies on the 5th line, where the user input is used directly in `snprintf()`.

2021 Q2f

Do flow-/path-/context-insensitive flow analysis for the snippet:

```c
int printf(untainted char * fmt, ...);
void read(int, tainted char * input, int);

char name[10];

read(0, name, sizeof(name));
char * x = name;
printf(x);
```

Give each assignment a qualifier:

- $\alpha =$ name
- $\beta =$ x

Now generate constraints:

- tainted $\leq \alpha$
- $\alpha \leq \beta$
- $\beta \leq$ untainted

$\alpha, \beta =$ untainted $\vee$ tainted

No valid solution! Therefore flow is illegal!

2021 Q2g
Do flow-sensitive and path-/context-insensitive analysis:

```c
int printf(untainted char * fmt, ...);
void read(int, tainted char * input, int);

char name[10];

read(0, name, sizeof(name));
char * x;
x = name;
x = "hello";
printf(x);
```

Give each assignment a qualifier:
- δ = name
- α = assignment of x to name
- β = assignment of x to "hello"

Now generate constraints:
- tainted $\leq \delta$
- $\delta \leq \alpha$
- untainted $\leq \beta$
- $\beta \leq$ untainted

Solution: $\alpha, \delta =$ tainted, $\beta =$ untainted.
Therefore flow is legal!

2021 Q4a
What is the vulnerability?

```c
int main(int argc, char ** argv) {
  if (argv[1]) return foo(argv[1]);
  else return foo(argv[0]);
}

int foo(char * arg) {
  char bar[128];
  if (sizeof(arg) > 128) {
    strcpy(bar, arg);
    exit(0);
  }
  strcpy(bar, arg);
  return strlen(bar);
}
```

Logic error on line 8 (will never fire as pointer size of arg will be 4 on 32-bit systems and 8 on 64-bit systems) which allows a buffer overflow on line 12.

8. January 2022 Paper

Q1a/b

```c
1  void MyFunc(char * str) {
2    char * name[3];
3    name[0] = "security";
4    name[1] = "engineering";
5    name[2] = NULL;
6  }
7
8  void foo(char * str) {
9    char buffer[100];
10   char arr[300] = "/sh/sh//";
11   strcat(arr, str);
12   MyFunc(str);
13 }
14
15 int main(int argc, char ** argv) {
16   if (!argv[1]) return -1;
17   foo(argv[1]);
18   exit(0);
19 }
```

Draw the stack of this program, incl. stack frames, local parameters, values. **[8]**
What is the value of %esp and %eip when foo() finishes execution (after epilogue)? **[4]**

Value	Origin
`argv`	main's caller
`argc`	
Return address from main	
Previous frame pointer	main()
`char* str` (argument to foo)	
Return address from foo (line 18)	
Previous frame pointer	foo()
`char buffer[100]`	
`char arr[300]`	
`char* arr` (argument to strcat)	
`char* str` (argument to strcat)	
`char* str` (argument to MyFunc)	
Return address from foo (line 13)	
Previous frame pointer	MyFunc()
`char* name[3]`	
Address of “security”	
Address of “engineering”	
NULL byte	

The value of `%esp` after foo's epilogue will be the previous frame pointer (main's frame pointer) which will, in this case, point to the return address from foo. The value of %eip will be pointing at “line 13”, i.e. the end of the function, since the return instruction has not been called just yet.

Q3

Perform taint analysis on snippet:

```c
int printf(untainted char * fmt, ...);
void read(int, tainted char * input, int);

char buffer[10];

read(0, buffer, sizeof(buffer));
char * x = buffer;
printf(x);
```

Let buffer be δ, let x be α.

- tainted $\leq \delta$
- $\delta \leq \alpha$
- $\alpha \leq$ untainted

No valid solution, therefore illegal flow!