Corrected

EPFL

1

Teacher: Prof. Dr. ETH Mathias Payer
CS-412 Software Security – Final Exam
28th May 2024
Duration: 120 minutes

Anon Ymous

SCIPER : **999999**

Do not turn the page before the start of the exam. This document is double-sided, has 20 pages, the last ones possibly blank. Do not unstaple.

- Place your student card on your table.
- **No other paper materials** are allowed to be used during the exam.
- Using a **calculator** or any electronic device is not permitted during the exam.
- For each multiple-choice question, mark the box(es) corresponding to the correct answer(s). Each question has **one or more** correct answers.
- For each multiple-choice question, we award:
  0 points if you give no answer,
  $\frac{2}{N}$ points per correctly checked or not checked answer, where N is the number of available responses (i.e., a maximum of 2 points).
  Each question has a minimum of 0 points, we do not award negative points.
- For each open-ended question we provide the awarded number of points next to the question.
- Only text **inside the marked boxes** will be graded for the open-ended questions.
- Use a **black or dark blue ballpen** and clearly erase with **correction fluid** if necessary.
- If a question is wrong, we may decide to nullify it.

Respectez les consignes suivantes | Observe this guidelines | Beachten Sie bitte die unten stehenden Richtlinien		
choisir une réponse | select an answer Antwort auswählen	ne PAS choisir une réponse | NOT select an answer NICHT Antwort auswählen	Corriger une réponse | Correct an answer Antwort korrigieren
ce qu'il ne faut **PAS** faire | what should **NOT** be done | was man **NICHT** tun sollte		

CORRECTED

First part: multiple choice questions

Question 1
Which of the following sanitizers occupy much more memory on 64-bit than on 32-bit platform?

- [x] AddressSanitizer.
- [x] MemorySanitizer.
- [ ] UndefinedBehaviorSanitizer.
- [x] LeakSanitizer.

Explanation:

Question 2
Which of the following statements about sanitizers are correct?

- [x] AddressSanitizer can detect out-of-bound reads and writes.
- [x] UndefinedBehaviorSanitizer is the only sanitizer that can be used in production.
- [ ] AddressSanitizer can be used as a mitigation against memory safety bugs.
- [x] ThreadSanitizer detects data races by instrumenting writes to global and heap variables and records which thread wrote the value last.

Explanation:

Question 3
Which of the following bugs may not manifest (i.e., cause user-observable behavior changes such as crashes) without deploying sanitizers? (Assuming Linux x86-64)

- [x] Stack buffer out-of-bound reads and writes.
- [x] Heap buffer out-of-bound reads and writes.
- [ ] NULL pointer dereferences.
- [x] Memory leaks.

Explanation:

Question 4
Which of the following statements about testing are correct?

- [x] Reaching definition is a data-flow analysis.
- [x] Assertions and sanitizers are both oracles in testing.
- [ ] Symbolic execution can only help find bugs on a provided concrete path.
- [x] Unit testing is a manual testing technique.

Explanation:

Question 5

You've decided to write your own testing tool to look for remote code execution vulnerabilities and after much work you come up with the following static analysis toolkit that you will sell for $200,000,000 per license:

```
#!/bin/sh
cat $1 | grep system
# cat $1 : dump file contents of argument to output
# grep system : print all lines containing the string "system"
# usage: ./analyzer.sh <filetoanalyze>
```

Which statements about your tool are correct?

- ☐ Your tool has no false negatives.
- ■ Findings by your tool could potentially be vulnerabilities.
- ■ Your tool will scale to huge code bases and would even work for Chrome.
- ☐ Analyzing your tool's findings can be easily automated.

Explanation: RCE can be achieved with other vulns (memory corruption) not only command injection. An instance of a use of system could potentially be vulnerable. The tool will be able to analyze many files/huge codebases, though quality of findings is questionable. Analyzing the tool's findings is the actual interesting challenge, which is a whole actual static analysis problem (data flow) which is not easy.

Question 6

You want to fuzz an obscure API function of an open-source JPG parsing library. So far you did not find any program using that API. What statements about setting up fuzzing for the library are correct?

- ☐ To get coverage the most efficient way is to use dynamic instrumentation on the compiled binary.
- ■ Using AFL++ as the fuzzer in this case has the disadvantage that you'd need to write your own client using that obscure API.
- ■ If the target API function interacts with a global library-internal variable, this variable needs to be reset in every iteration to ensure crashes are reproducible in libFuzzer.
- ☐ Bugs found during this fuzzing campaign will directly result in real-world vulnerabilities.

Explanation: Source code is available -> no performance penalty using source code instrumentation. Afl++ needs a library client/driver. Libfuzzer does persistent fuzzing so this state will be persisted across fuzzing iterations. Not necessarily the API may not be used in real programs or the data fed into this API not attacker-controlled.

CORRECTED

Question 7

Your brand-new coverage-guided greybox fuzzer writes the coverage instrumentation into a `COVSIZE`-sized `char` array `covmap`. The instrumentation pseudo-code inserted at each basic block:

```
long rip = get_rip(); // retrieve current instruction pointer
covmap[rip % COVSIZE]++;
```

Which of the following statements about your coverage collection instrumenation are correct?

- [ ] The type of this coverage instrumentation is edge coverage.
- [x] The type of this coverage instrumentation is block coverage.
- [ ] The type of this coverage instrumentation is path coverage.
- [ ] The type of this coverage instrumentation is function coverage.
- [x] Limiting `COVSIZE` to a small number (for example 0x100) may lead to collisions.
- [ ] There are no downsides to increasing `COVSIZE` to a large number (for example 0x100000).

Explanation: Type of coverage is block coverage, only current basic block is taken into account. Increasing COVSIZE makes parsing the coverage map more costly. Smaller COVSIZE makes it more likely different PCs end up in same index of coverage map.

Question 8

In 2023, a heap buffer overflow vulnerability was found in libwebp, a popular, heavily-fuzzed image parsing library by Google. Google's massive fuzzing campaigns had failed to find this vulnerability. To trigger the vulnerability a specific sequence of about 20 bytes is required. Changing a byte in this sequence does not change basic blocks and edges traversed.
Which of the following statements about fuzzing to find this vulnerability are correct?

- [ ] The vulnerability remained undetected due to Google not using AddressSanitizer.
- [x] Using edge coverage the only way a fuzzer can discover the crash is by guessing the crashing 20-byte sequence, with a probability of $\frac{1}{2^{160}}$.
- [ ] The crash is more likely to be discovered using block coverage than edge coverage.
- [ ] The crash can not be found by seeding the fuzzer with inputs that only differ with the crashing input by one byte.

Explanation: AddressSanitizer or not the issue of not finding the crash is having to guess 20 bytes. Block or Edge don't make a difference in this case. No this is how the crash can be found since the fuzzer only needs to guess one byte (and it's location).

Question 9

Which of the following statements about program testing/analysis are correct?

- [ ] Manual testing is always whitebox.
- [x] Compile time warnings are a lightweight static analysis.
- [ ] False positives generally have a higher negative impact on security than false negatives.
- [ ] Formal verification can scale to large codebases with around 100,000,000 lines of code.

Explanation: Manual testing can be everything. Depends on the use case, cannot be generalized. Very hard to make work even on small code bases.

Corrected

Question 10
Which of the following statements about software daemons are correct?

- [x] Daemons are long running, complex, and exposed.
- [x] Compartmentalization helps reduce attack surface.
- [ ] Daemons are easier to protect as per-user contexts are always separate.
- [ ] ASLR protects daemons against information leaks.

Explanation:

Question 11
Cross-Site Scripting is a key attack vector that can be carried out through the following means:

- [x] By storing the malicious JavaScript on the server.
- [x] By storing the malicious JavaScript in the URL.
- [x] By storing the malicious JavaScript in an object that is unpacked by client-side JavaScript.
- [ ] By storing the malicious JavaScript in a browser extension.

Explanation:

Question 12
SQL injection can be mitigated through:

- [x] validating the inputs (i.e., allow listing possible inputs)
- [x] escaping the inputs (i.e., escaping special characters)
- [x] prepared SQL statements (i.e., leveraging a better API)
- [ ] reducing privileges of the SQL server (i.e., not running it as root)

Explanation:

Question 13
What are properties of the following bug:

```
<html><head><title>Display a file</title></head>
<body>
<? echo system("cat␣".$_GET['file']); ?>
</body></html>
```

- [x] This is a command injection bug.
- [x] This bug type is often present in routers.
- [ ] This bug is best mitigated by filtering `;` and `|`.
- [ ] Forking a separate process is necessary to mitigate this bug.

Explanation:

CORRECTED

Question 14
External dependencies (i.e., included libraries) can become insecure if vulnerabilities are discovered in them. In this context, which statements are true.

- [x] A developer must carefully document all external dependencies.
- [x] It may make sense to implement simple functionality to "internalize" the responsibility and reduce reliance on external code.
- [ ] External dependencies can be updated without needing to consider internal dependencies or used functionality.
- [ ] Staying on an older version is safe as long as no bugs are disclosed in that version.

Explanation:

Question 15
The `chroot` syscall creates a filesystem-based sandbox. When chroot is called on a folder, a process spawned inside the chroot container cannot access any file outside that folder. However, this is widely considered a very weak form of sandboxing.
Imagine you are a process with root privileges inside a chroot sandbox. Auxiliary filesystems such as `/dev`, `/proc`, `/sys` are available in the sandbox. How do you escape? Select all potential escape routes.

- [x] You can use the `mount` syscall to remount the rest of the disk in a subfolder you can access.
- [x] Processes are still visible from inside the chroot. You can ptrace a process outside the chroot and write your shellcode there to escape.
- [ ] Any new processes spawned from inside the chroot sandbox are not restricted by it. So spawning a new shell will be enough to be able to access files outside the sandbox.
- [ ] As root, you can disable chroot system-wide through a dedicated system call.

Explanation: Chroot is only at the filesystem level. Ptracing other processes and spawning shells in their environment works. The mount syscall also works, as you can mount "/" into whatever subfolder and access the whole disk. New processes are always in the same chroot sandbox of the parent.

Question 16
PAC (Pointer Authentication Codes) is a new Code Pointer Integrity mechanism for modern ARM devices (aarch8.3 and upwards). It works by using the upper bits of a pointer to store a cryptographically secure signature of the pointer. Usually, only the first 48 bits of a pointer are used; the PAC signature is stored on the upper 16 bits remaining. Every time a pointer is dereferenced, a special instruction checks if the signature is still valid (i.e., the pointer has not been tampered with). What are the advantages of such a mechanism compared to more traditional forward-edge Control-Flow Integrity methods?

- [x] You can protect backwards edges too, by signing the return address before pushing it on the stack.
- [ ] PAC is able to prevent stack overflows from vector instructions.
- [ ] You can use it to prevent address leaks, masking the location of libc or the PIE base.
- [ ] The signature can be crafted in such a way that bits can be reused to store intermediate results of floating point computations.

Explanation:

Corrected

Question 17

You found a stack buffer overflow in a program that gives you complete control of the stack contents. You use it to overwrite the return address to start a ROP chain. Select all the answers that could be used as the last gadgets in a ROP chain in which your goal is to jump to 0x41414141.

- [x] `xor rax, rax; mov rax, 0x41414141; call rax`
- [x] `xor rax, rax; mov rax, 0x41414142; ret`
- [x] `mov rax, 0x41414141; push rax; ret`
- [x] `push 0x41414141; push rsp; add rsp, 8; ret`
- [ ] `push 0x41414141; push rsp; xor rsp, rsp; ret`

Explanation:

Question 18

Consider the following code:

```
1 int main() {
2     char input[512];
3     gets(input);
4     printf(input);
5 }
```

Assume the mitigations present are: ASLR, DEP, stack canaries. PIE is not enabled. No other libraries other than libc are linked with it. How tragic is this code? Select all answers that apply.

- [x] You can achieve Control-Flow Hijacking.
- [x] You can achieve Arbitrary Read Anywhere.
- [x] You can achieve Arbitrary Write Anywhere.
- [x] You can achieve Arbitrary Code Execution.

Explanation:

Question 19

In which of the following scenarios is it most sensible to use seccomp filters?

- [ ] When you need to enforce mandatory access control policies across the entire system.
- [x] When you want to restrict the system calls available to a specific process to enhance its security.
- [ ] When you need to implement user authentication and authorization mechanisms.
- [ ] When you want to monitor and log system activity for auditing purposes.

Explanation:

CORRECTED

Question 20

In 2001, Michael "MaXX" Kaempf published one of the most influential articles in hacker culture, called *"Vudo - An object superstitiously believed to embody magical powers"*. The article was published in the *Phrack* zine, together with another very famous article *"Once Upon A Free()"*, initiating a lot of young hackers to the art of heap exploitation.

MaXX showcased the wonders of heap exploitation by publishing a massive exploit on the `sudo` binary through command line flags.

Here's the code of the vulnerable function `do_syslog`. The `char* msg` is user-controlled and comes from the argument list (flags) passed to the sudo binary.

Can you find the bug yourself?

```
1  /*
2   * Log a message to syslog, pre-pending the username and splitting the
3   * message into parts if it is longer than MAXSYSLOGLEN.
4   */
5  static void do_syslog( int pri, char * msg )
6  {
7      int count;
8      char * p, * tmp;
9      char save;
10
11     /*
12      * Log the full line, breaking into multiple syslog(3) calls if
13      * necessary
14      */
15 [1] for ( p=msg, count=0; count < strlen(msg)/MAXSYSLOGLEN + 1; count++ ) {
16 [2]     if ( strlen(p) > MAXSYSLOGLEN ) {
17             /*
18              * Break up the line into what will fit on one syslog(3) line
19              * Try to break on a word boundary if possible.
20              */
21 [3]         for ( tmp = p + MAXSYSLOGLEN; tmp > p && *tmp != ' '; tmp-- ) ;
22 [4]         if ( tmp <= p )
23                  tmp = p + MAXSYSLOGLEN;
24
25             /* NULL terminate line, but save the char to restore later */
26             save = *tmp;
27 [5]         *tmp = '\0';
28
29             if ( count == 0 )
30                 SYSLOG( pri, "%8.8s : %s", user_name, p );
31             else
32                 SYSLOG( pri,"%8.8s : (command continued) %s",user_name,p );
33
34             /* restore saved character */
35 [6]         *tmp = save;
36
37             /* Eliminate leading whitespace */
38 [7]         for ( p = tmp; *p != ' '; p++ ) ;
39 [8]     } else {
40             if ( count == 0 )
41                 SYSLOG( pri, "%8.8s : %s", user_name, p );
42             else
43                 SYSLOG( pri,"%8.8s : (command continued) %s",user_name,p );
44         }
45     }
46 }
```

- ☐ [1]: The for loop can go over past the end of the string.
- ☐ [2]: strlen can go over past the end of the string
- ■ [7]: The for loop does not check for null byte string termination.
- ☐ [4]: The check should be `if ( tmp > p )`

Explanation:

CORRECTED

Question 21
On a modern x86-64 Linux system compiled with default compiler settings, ASLR...

- [x] ... changes the start of the data segment every time the program is run.
- [ ] ... prevents sharing the memory allocated for the text segment of shared libraries opened by different processes.
- [ ] ... can change the address of a segment to any address.
- [x] ... probabilistically mitigates what an attacker can do with only an arbitrary write primitive.

Question 22
Software-based implementations of shadow stacks...

- [x] ... can contain frame pointers.
- [ ] ... contain a full copy of the stack frames.
- [x] ... incur a performance/memory penalty regardless of whether there are opportunities for data corruption or not.
- [ ] ... ensure that none of the variables are present in both stacks simultaneously.

Question 23
Control flow integrity (CFI) ...

- [x] ... is the strongest existing/deployed mitigation to protect the forward edges of control flow.
- [ ] ... is the strongest existing/deployed mitigation to protect the backward edges of control flow.
- [ ] ... has a prohibitively high performance cost (>5%).
- [x] ... does not completely prevent control-flow hijacking.

Question 24
Assume you have valuable data stored locally on your laptop's internal disk (i.e., cloud storage and similar are not of concern for this question). Under which of the following threat models is your locally stored data at risk of being leaked to a malicious actor? Note that Secure Boot ensures that the boot sequence cannot be tampered with.

- [x] An attacker has physical access to your laptop and your data partition is encrypted but Secure Boot is turned off.
- [ ] An attacker has physical access to your laptop and you employ full disk encryption as well as Secure Boot.
- [x] An attacker managed to install a Trojan on your system.
- [ ] An attacker managed to make you click on a phishing link.

Explanation: Only data partition encrypted and no Secure Boot allows an attacker to interfere with the boot process and run malware that exfiltrates your data as soon as it's decrypted. A trojan accessing your filesystem can also exfiltrate data. Clicking on a phishing link per se does not leak your local data.

Corrected

Question 25

In order to apply the principle of least privilege to your flatshare's shared fridge,...

- ☐ you attach a lock to the fridge door that only you have the key to.
- ☐ you poison some of your food.
- ☐ you attach a lock to the fridge door that every flatmate has the key to.
- ■ you provide every flatmate with a dedicated, locked compartment in the fridge that only they have the key to.
- ☐ you give up on fridge sharing and provide every flatmate with their own fridge in their room.

Explanation: The wrong answers all either reduce functionality (e.g., giving up on sharing the fridge violates the property of the shared fridge) or provide somebody with too many privileges.

Question 26

You're a software engineer and recently released the software you've been working on with your team for the past two years to the public. During development, you paid attention to security best practices according to the secure software development lifecycle. What tasks do you still need to fulfill after publication of the software to keep your good security track record?

- ■ Regularly update all third-party components
- ■ Track security advisories for third-party components
- ■ Re-test and review the software's compliance with defined security requirements after every update
- ☐ HTTPS is necessary to securely deploy updates
- ■ Deploy updates securely, i.e., sign your updates
- ☐ Re-implement third-party components with security flaws in-house in a memory-safe language

Explanation: Security advisories should be taken seriously and the affected components updated accordingly. The software should of course also be tested against regressions on updates. HTTPS on its own is not a mechanism for secure software distribution. If the distribution is correctly signed (and the signature verified, of course), even downloads over insecure channels are secure (e.g., some Linux distros still using HTTP for package repositories). Reimplementing third-party components in memory-safe languages is neither part of the secure software development lifecycle nor a guarantee for the absence of security flaws (potential logic bugs could still lead to exploitable software, no matter whether the component is written in a memory-safe language or not).

Corrected

Question 27

Consider the following code snippet. Which attack primitive can an attacker gain in which line?

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4      char name[32] = { 0 };
5      int age = 0;
6      printf("Please provide your name:\n");
7      scanf("%s", name);
8      printf("Please provide your age:\n");
9      scanf("%d", &age);
10     printf("Your name is: ");
11     printf(name);
12     printf("Your age is: %d\n", age);
13     return 0;
14 }
```

- ☐ Line 7: arbitrary read
- ☐ Line 7: arbitrary write
- ■ Line 7: out of bounds access
- ☐ Line 9: arbitrary read
- ☐ Line 9: arbitrary write
- ☐ Line 9: out of bounds access
- ■ Line 11: arbitrary read
- ■ Line 11: arbitrary write
- ■ Line 11: out of bounds access

Explanation: The `scanf` in line 7 can cause a buffer overflow because the input string length is not bounded. The `scanf` in line 9 is fine. The `printf` in line 11 is a classic format string vulnerability, providing both arbitrary read and write primitives.

Question 28

Your friend is designing a challenge for a CTF. A player is provided with a terminal interface as a non-root user. The flag is found in the file /flag, owned by root, not executable by any user, and only readable/writable by root. In the intended solution, a player should leverage a bug in the Linux kernel. What are the attacker goal(s) intended by your friend?

- ☐ Denial of Service
- ■ Leak information
- ■ Privilege escalation
- ☐ None of the others

Explanation: becoming root -> privilege escalation
reading the flag -> unintended transfer of sensitive information to the attacker -> leak information

CORRECTED

Question 29

After learning that you can win millions of dollars at the pwn2own competition by exploiting real-world software, you decide to use the technical skills acquired in the software security class to target the v8 JavaScript engine, which is written in C++. While you first expected this to be particularly difficult, you then learn that the engine has a RWX mapping for WASM code, and you can execute code that is mapped there! Select the correct affirmations among the following:

- ☐ No additional vulnerability or bug is needed to achieve code execution
- ■ Assuming that you have arbitrary write and that you know the address of the RWX mapping, you can target such mapping for a code corruption or a code injection attack
- ☐ Assuming that you have arbitrary write and that you know the address of the RWX mapping, you can only rely on attacks based on code reuse
- ☐ Assuming that you have arbitrary write and that you know the address of the RWX mapping, it is impossible to achieve code execution

Explanation:

- a RWX mapping with code you can execute is not enough to achieve code execution, one would need to at least be able to inject or corrupt code in this mapping
- with arbitrary write and if you know the RWX mapping's location, you can corrupt/inject code in that mapping, then execute it
- by the above point, you are not forced to rely on code reuse attacks
- in the absence of RWX mappings, some vulnerabilities still allow for control flow highjack based on code reuse attacks for example

Question 30

Your friend, concerned about writing memory safe and type safe code, is implementing their own heap allocator (malloc,free...). Assume that the allocator keeps track of the pointers passed to free in a linked list called *free_list*. When a pointer is freed, it is compared to the pointer that is stored at the head of the *free_list* (if any), if they are equal, execution is aborted. Otherwise, it is inserted at the head of the *free_list*. Assume that execution is aborted whenever a pointer that was not first returned by malloc is passed to free. Assume that the integrity of the *free_list* is guaranteed. What kind(s) of safety (if any) are then guaranteed on *ptr* during a call to *free(ptr)* thanks to the introduced checks?

- ☐ Full spatial memory safety
- ☐ Full temporal memory safety
- ☐ Full type safety
- ■ None among the other answers

Explanation: The only safety this mitigation is related to is temporal memory safety. However, it only compares *ptr* against the head, and not with all pointers in the free_list, thus double free are still possible and temporal safety of ptr during the free call is not guaranteed.

Second part: open-ended questions

Answer in the empty space below each question. Your answer should be carefully justified, and all the steps of your argument should be discussed in details. Leave the check-boxes empty, they are used for the grading.

Question 31: *This question is worth 15 points.*

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7
☐8 ☐9 ☐10 ☐11 ☐12 ☐13 ☐14 ■15

You got hired as a software engineer to work for a Software-as-a-Service (SaaS) startup. You're part of a team that tests experimental technologies to improve the security of the company's SaaS offerings.
The company offers hosted software that is presented to the end user as a web app. On the server side of the application, the software interacts with a database, an in-memory cache to hide database latency from end users, and has to conduct heavyweight analysis of the user-provided data before storing the results back into the database. Because the company is of course trying to keep up with technological trends, the software also includes LLM-supported user data entry.
The company has already grown pretty big over the past years but is still in a kind of "move-fast-and-break-things" phase. Therefore, the original sysadmin (singular, all the resources went into frontend devs) just deployed the server-side backend as a huge monolithic blob with a webserver onto a dedicated Linux machine, installed a database, the aforementioned caching software, and a few GPUs for LLM inferencing.
Because you've heard of concepts like principle of least privilege and fault compartments, you're eager to revamp this infrastructure and introduce a bit more security into this setup.
You are considering containerization, virtualization, microkernels. Pick and discuss **two** of these practical approaches for introducing security boundaries and clear fault compartments in this setup. Discuss trade-offs of the chosen approaches by listing and explaining two advantages and disadvantages each per approach.

Question 32: *This question is worth 15 points.*

Consider the following code snippet:

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
    return;
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr != NULL) {
        printf("Enter %d elements:\n", n);
        for (int i = 0; i < n; i++) {
            scanf("%d", &arr[i]);
        }
        quick_sort(arr, 0, n - 1);
        printf("Sorted array: \n");
        for (int i = 0; i < n; i++)
            printf("%d ", arr[i]);
        free(arr);
        return 0;
    }
    return 1;
}
```

(a) **Inside** function `quick_sort`, how many instrumentation points are required to collect edge coverage information? How many edges are there? Ignore compiler optimization like inlining. Hint: drawing the CFG may help.

(b) **Inside** function `partition`, how many instrumentation points are required to collect edge coverage information? How many edges are there? Ignore compiler optimization like inlining. Hint: drawing the CFG may help.

(c) The code is instrumented with AddressSanitizer (ASan). In function `swap`, what checks does ASan do? Explain them with giving information about the concrete locations.

(d) What bugs of heap objects can ASan detect? What functionality does the ASan instrumentation code provide to detect them?

CORRECTED

Question 33: *This question is worth 15 points.*

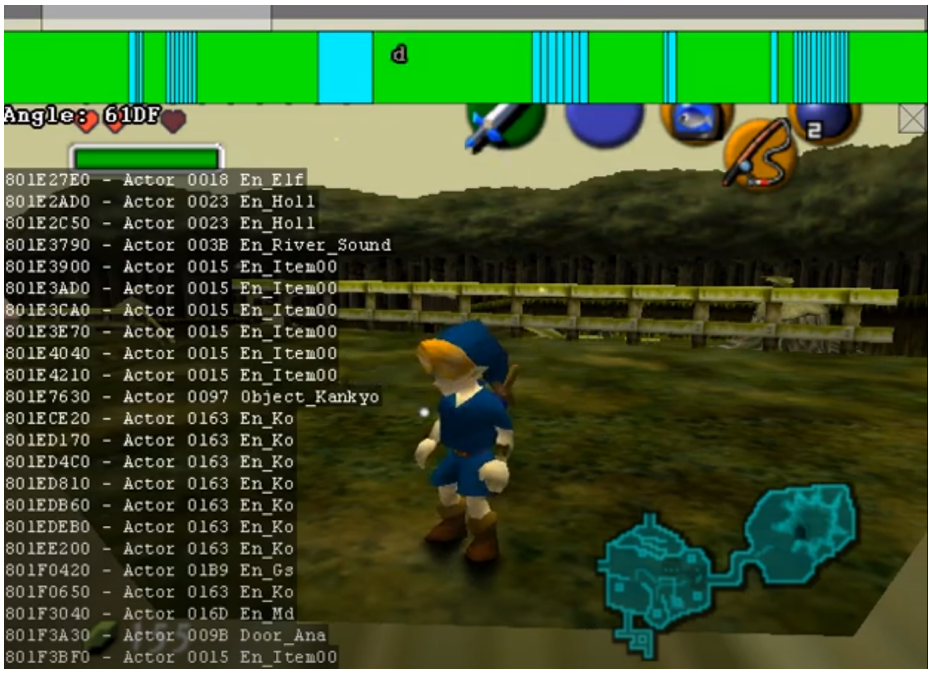
Google rolls out a new Android version each year and increases the API version alongside. Each API version brings new functionalities and security improvements, deprecating older API calls as well. Under the scenario, consider the following questions. The Android baseline system only has very few system libraries installed, providing basic functionality.

(a) What are the security trade-offs for choosing a given minimal API level for developing an app? List three reasons why going for lower API levels impacts the ecosystem

(b) Many apps ship with individual copies for native libraries. List one key security downside and benefit.

(c) Why does Google control the app market? List 3 reasons and state if they are positive or negative for the user

Question 34: *This question is worth 15 points.*



According to the website `zeldaspeedruns.com`, in the N64 masterpiece "Zelda: Ocarina of Time" there is a way to achieve Arbitrary Code Execution from inside the game. This is the explanation that the website provides:

[...] The heap is a doubly linked list of actor instances (the variables for an actor, such as its position), actor overlays (code used by actors), and more. Some actors have references to other actors on the heap. For example, Link stores a reference to the actor he is holding above his head, such as a bush, rock, bomb, or bombchu. The Boomerang stores a reference to the actor it is carrying, such as a Rupee, Recovery Heart, or Gold Skulltula Token. When Link or the Boomerang move, they change the position of the actors they are carrying, as well as some other data.
Under specific circumstances, it is possible to unload the actor being carried, then load a new actor instance or overlay in its place. The actor carrying that actor will continue to write values such as position data to that location, which could now be an entirely different actor or even actor code. The act of writing that data to the referenced location, despite the reference no longer pointing to the original actor, is known as Stale Reference Manipulation.
Typically, several actors and rooms are carefully loaded in a particular order, causing a specific part of an actor instance or overlay to align with the reference to the region in memory which previously contained the carried actor. Manipulating memory in this way is known as heap manipulation.

The website then continues on with the technical details on how ACE is obtained from there.

- The speedrunners call this vulnerability *Stale Reference Manipulation* — however, this is actually an instance of an very well known heap vulnerability that hackers have been exploiting since malloc was ever implemented. Which one?
- Which mechanism (mitigation or sanitizer) could be used to prevent (or detect) this bug class? Either name an existing one or propose a modification of an existing one. Make sure the mitigation *always* works (i.e., 95% of the times is not enough).

Corrected

Corrected