Teacher: Prof. Dr. ETH Mathias Payer
CS-412 Software Security – Final Exam
30th May 2023
Duration: 120 minutes

1

Anon Ymous

SCIPER : **999999**

Do not turn the page before the start of the exam. This document is double-sided, has 16 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:
  2 points by default,
  0 points if you give no answer,
  $-\frac{2}{3}$ point per incorrectly checked or missed answer.
  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.
- 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

▨ ☒ ○ ⊡ ▨ ☐

First part: multiple choice questions

Question 1

Select all of the regions that ASLR randomizes the address when PIE is not enabled in x86-64 Linux:

- ☐ The heap
- ☐ The gdb server stub
- ☐ The libraries
- ☐ The stack
- ☐ The executable

Question 2

Which of the following statements about code instrumentation is/are correct?

- ☐ Binary rewriting-based coverage collection has lower runtime overheads than compiler-based instrumentation.
- ☐ We should instrument basic blocks when collecting edge coverage.
- ☐ The instrumentation code for coverage collection should not change the original functionality.
- ☐ We can only do binary rewriting on position-independent code (PIC).

Question 3

Which of the below security policies are violated by the following code snippet?

```c
char *foo(int b) {
    char c = (char)(b & 0xff);
    return &c;
}


int main(int argc, char *argv[]) {
    char buf[100] = {0};
    int x = 0x539; // 1337
    int y = (int)(*foo(x));
    putc(buf[y]);
    return 0;
}
```

- ☐ Type safety
- ☐ Spatial memory safety
- ☐ None of the other answers
- ☐ Temporal memory safety

Question 4

What is/are the goal/s of compartmentalization?

- ☐ Make faults more severe as the surrounding code is smaller.
- ☐ Better performance (i.e., lower overhead) since a compartment can fail without affecting others.
- ☐ Allow easier abstraction of functionalities across components.
- ☐ Isolate faults to individual (ideally small) components.

Question 5
Which of the following statements about mitigations are correct?

☐ Shadow stacks can be implemented in software with zero overhead.
☐ Safe stacks protect against corruption of all data on the stack.
☐ Code-Pointer Integrity (specifically the implementation described in the slides) uses a separate stack to protect code pointers.
☐ Control-Flow Integrity can efficiently protect the forward edge but, when using target sets, is limited on the backward edge

Question 6
Which of the following statements about libFuzzer is/are correct?

☐ Unit tests may serve as foundation to create libFuzzer fuzzing stubs.
☐ It is better to put narrow targets into the fuzzing stubs, e.g., if a target can parse several data formats, split it into several targets, one per format.
☐ In libFuzzer's default mode (not fork-mode), the tested APIs must not contain `exit()`.
☐ libFuzzer can only test single-threaded targets.

Question 7
Current software is complex and often relies on external dependencies. What are the security implications?

☐ Closed source code is more secure than open source code as it prohibits other people from finding security bugs.
☐ During the requirement phase of the secure development lifecycle, a developer must list all the required dependencies.
☐ It is necessary to extensively security test every executable on a system before putting it in production.
☐ As most third party software is open source, it is safe by default since many people reviewed it.

Question 8
Which of the following statements about testing is/are correct?

☐ Tests prove the presence of bugs but not their absence.
☐ Tests prove the absence of bugs but not their presence.
☐ In static analysis, determining the correct target set of indirect calls is typically challenging but required to minimize over-approximation of targets.
☐ Concolic execution specializes in finding bugs close to the path of the provided concrete input.
☐ Compared to dynamic analysis, static analysis is more susceptible to state space explosion.

Question 9
Which of the following attack vectors apply to mobile Android systems?

☐ Overprivileged apps may be abused as a confused deputy, allowing malicious apps to steal access to their privileges.
☐ Hardware vendors like $am$ung are primarily interested in making money and not in providing software updates, resulting in outdated software that is vulnerable to attacks.
☐ Malicious apps can intercept network traffic of benign apps.
☐ Apps may maliciously declare intent filters to receive intents from benign apps.

Question 10
Which of the following statements about fuzzing is/are correct?

- ☐ Greybox fuzzing is always the better alternative to blackbox fuzzing.
- ☐ Generational fuzzing requires more manual work (to specify the generator policies) than mutational fuzzing, but can generate high-quality seeds.
- ☐ Greybox fuzzing keeps track of concrete program paths to abstract behavior.
- ☐ Blackbox fuzzers can make use of initial seeds.

Question 11
For security reasons, you accept the performance and memory overhead introduced by common sanitizers and deploy them in your user-facing production server software. Assuming that all memory safety bugs in your software are detected by the sanitizers, which of the following properties do the sanitizers provide to your code?

- ☐ Confidentiality of the program data
- ☐ Accountability of accesses to the program
- ☐ Availability of the program
- ☐ Integrity of the program data

Question 12
In x86-64 Linux, the canary is **always** different for every?

- ☐ Process
- ☐ Function
- ☐ Thread
- ☐ Namespace

Question 13
Which of the following statements are true about command injection?

- ☐ To mitigate command injection, it is best to replace powerful shell commands (e.g., `system()`) with less privileged alternatives such as `read_file()`.
- ☐ Command injection can be mitigated, in general, by prohibiting the `";"` character.
- ☐ Command injection allows the attacker to launch new processes or invoke internal shell commands.
- ☐ The root cause of command injection is the lack of distinction between data plane and control/code plane.
- ☐ Command injection is unique to PHP web applications.

Question 14
What makes `C++` inherently NOT type safe (i.e., unsafe casts may cause an object of type X be interpreted as type Y even though types X and Y are not related)?

- ☐ The absence of type information (and checks) for non-polymorphic objects at runtime.
- ☐ Class and struct are indistinguishable at run-time due to their memory layout.
- ☐ The polymorphic inheritance between interface and classes.
- ☐ The use of function pointers makes static analysis intractable.

Question 15
For which kind of bugs does default LLVM provide sanitizers?

☐ Race conditions between threads
☐ Buffer overflows
☐ Memory leaks
☐ Logic bugs

Question 16
Does AddressSanitizer prevent **all** use-after-free bugs?

☐ No, because UAF detection is not part of ASan's feature set.
☐ Yes, because free'd memory is unmapped and accesses therefore cause segmentation faults.
☐ Yes, because free'd memory chunks are poisoned.
☐ No, because quarantining free'd memory chunks forever prevents legit memory reuse and could potentially lead to out-of-memory situations.

Question 17
Which defense(s) highlight the principle of least privilege in software security?

☐ A stack canary because it will signal any stack-based attack.
☐ DEP bits by disallowing execution on certain memory pages because code is restricted to code pages.
☐ CFI protection on the forward edge because the check limits reachable targets.
☐ Applying updates regularly because software updates always reduce privileges.

Question 18
Daemons are just long running processes. When applying mitigations to these processes, several aspects change. Which ones?

☐ DEP becomes less effective as compiler optimizations are turned on, allowing the attacker to inject new code.
☐ ASLR becomes less effective as multiple requests across different users are handled in a single process.
☐ Stack canaries become less effective as multiple requests are handled by the same thread.
☐ CFI becomes less effective as the concurrent clients cause more targets to be available.

Question 19
Which of the following statements about coverage-guided fuzzing is/are correct?

☐ Due to the coverage feedback, a small random perturbation of a seed can have a significant impact on further exploration.
☐ Fuzzers that have higher code coverage always find more bugs.
☐ Counting the number of times the covered code has been executed provides a more fine-grained view of program behavior than only "covered/not covered" binary code coverage.
☐ Redundant seeds in the corpus will reduce fuzzing efficiency.

Question 20
Which of the following hold(s) true about update deployment in the secure development lifecycle?

- ☐ Updates may bring new code that may be buggy, so additional monitoring is required after deploying an update.
- ☐ One motivation for automatic updates is for manufacturers to ensure that users have the latest code installed.
- ☐ You should always deploy third party updates automatically and immediately in your project.
- ☐ Not allowing rolling back to previous versions is necessary in the Secure Development Lifecycle.

Question 21
Which of the following measures will always improve fuzzing executions per second?

- ☐ Reducing overheads imposed by the fuzzing framework.
- ☐ Performing structure-aware input generation.
- ☐ Collecting code coverage as feedback.
- ☐ Providing dictionaries for input generation.

Question 22
Which of the following hold true for cross-site scripting (XSS)?

- ☐ XSS can only be used to leak private data of a user.
- ☐ Reflected XSS requires that the server stores the injected code but the user does not need to click on any special link.
- ☐ XSS is a form of code injection that gives the attacker arbitrary code execution.
- ☐ Client-side XSS is a unique problem of GMail.

Question 23
Which of the following in Linux x86-64 assembly snippets can be used as a gadget AND can be chained with more gadgets (e.g., in a ROP/JOP chain)?

- ☐ `xor rbx, rbx; xor rbx, -1; push rbx; ret`
- ☐ `mov eax, -1; call rax`
- ☐ `pop rbx; pop rax; ret`
- ☐ `pop rbx; pop rax; jmp rax`

Question 24
Which of the following apply to recent Android-based mobile systems but not to Linux-based desktop systems?

- ☐ All apps run in a strict container with only limited system calls available.
- ☐ Apps should use the binder interface to communicate with other apps.
- ☐ Arbitrary apps can exchange files through shared directories.
- ☐ By default, each app runs as its own user.

Question 25

Given this program snippet which is part of a large (> 10000 LoC) codebase, which of these statements are true, given that the contents of string "s" are attacker controlled, the attacker can run the function f only once, the attacker has access to the binary and the binary is compiled for x86_64 on a modern Linux system?

```c
#include <string.h>
void f(char* s) {
    char b[100] = {0};
    memcpy(b, s, strlen(s));
    printf("\%s", b);
}
```

☐ If this program is compiled with no mitigations, an attacker can gain remote code execution.

☐ If this program is compiled with DEP (Data-Execution Prevention) and no other mitigation, an attacker can gain remote code execution.

☐ If this program is compiled with stack canaries and no other mitigation, an attacker can leak the canary.

☐ If this program is compiled with stack canaries and no other mitigation, an attacker can reliably gain remote code execution.

Question 26

Which of the following statements about symbolic execution is/are correct?

☐ Symbolic execution requires actually running the target program.

☐ Symbolic execution can efficiently handle and solve constraints in programs with simple logics but large input space.

☐ State space explosion is a common challenge for symbolic execution.

☐ Symbolic execution can always accurately model a system's environment (e.g., system calls, file I/O, and network I/O).

Question 27

Which of AddressSanitizer (ASan), MemorySanitizer (MemSan), UndefinedBehaviorSanitizer (UBSan) or ThreadSanitizer (TSan) can detect bugs (if any) in the following code snippet?

```c
int factorial(int x) {
    // Calculate and return the factorial of a number
    int result = 1;
    for (int i = 1; i <= x; i++) {
        result *= i;
    }
    return result;
}

int main(int argc, char *argv[]) {
    printf("%d\n", factorial(32));
    return 0;
}
```

☐ ASan

☐ UBSan

☐ MemSan

☐ TSan

☐ There are no bugs in the snippet.

☐ There is at least one bug in the snippet, but none of the mentioned sanitizers can detect it.

Question 28

Which of AddressSanitizer (ASan), MemorySanitizer (MemSan), UndefinedBehaviorSanitizer (UBSan) or ThreadSanitizer (TSan) can detect bugs (if any) in the following code snippet?

```c
int sum_up_to(int x) {}
    // Return sum of integers up to x
    int result = x;
    for (int i = x; i >= 0; i--) {
        if (INT_MAX - i <= result) {
            break;
        }
        result += i;
    }
    return result;
}
```

- ☐ ASan
- ☐ TSan
- ☐ There are no bugs in the snippet.
- ☐ MemSan
- ☐ There is at least one bug in the snippet, but none of the mentioned sanitizers can detect it.
- ☐ UBSan

Question 29

Which of AddressSanitizer (ASan), MemorySanitizer (MemSan), UndefinedBehaviorSanitizer (UBSan) or ThreadSanitizer (TSan) can detect bugs (if any) in the following code snippet?

```c
int sum_array(int *arr, size_t len) {
    // Return sum of array elements
    int result = 0;
    for (size_t i = 0; i <= len; i++) {
        result += arr[i];
    }
    return result;
}
```

- ☐ UBSan
- ☐ TSan
- ☐ ASan
- ☐ There is at least one bug in the snippet, but none of the mentioned sanitizers can detect it.
- ☐ There are no bugs in the snippet.
- ☐ MemSan

Question 30
Which of AddressSanitizer (ASan), MemorySanitizer (MemSan), UndefinedBehaviorSanitizer (UBSan) or ThreadSanitizer (TSan) can detect bugs (if any) in the following code snippet?

```c
int min_array(int *arr, size_t len) {
    // Return the minimum value in an array
    int result;
    for (size_t i = 0; i < len; i++) {
        if (arr[i] < result) {
            result = arr[i];
        }
    }
    return result;
}
```

☐ There is at least one bug in the snippet, but none of the mentioned sanitizers can detect it.
☐ MemSan
☐ There are no bugs in the snippet.
☐ TSan
☐ ASan
☐ UBSan

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 10 points.*

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9 ☐10

You got hired as a software engineer right after the software security lecture to work in a bank desperate for talent. Your first mission is to implement some extra features on top of the customer database to handle new privacy regulations forced by the EU coming into effect in three months. The implementation of these features is time-critical and will fill up almost all your time.
You realize that the database was developed in-house in C 40 years ago (a few 10'000s LoC) and is running without any mitigations as it was compiled with an ancient version of GCC running on a Linux virtual machine from 2000. The other developers did not take any software security classes and were purely focused on implementing features. Your boss tells you not to worry about code quality of this database engine as the availability is good (>99.9%) and the bankers using the system really do not want instability.
Even if it is not part of your task, you are still worried and schedule a discussion with your boss to speak about the security implications of the current implementation of the database.
During your preparation, you come up with **two** practical and acceptable ideas for improving the security of the database (e.g.,unrelated to your primary task of implementing new features) that could be implemented in the three next months in your spare time. What are they? What is their advantage? How would you argue for them? Discuss trade-offs and reasons why these ideas are feasible compared to alternatives.

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

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

You're a compiler engineer in a software company and you are tasked to develop a compiler for your company's new architecture `RISC-X` from scratch. `RISC-X` uses a 64-bit virtual address space similar to x86-64. Because of the popularity of sanitizers on other platforms, you intend to develop support in your toolchain to detect out-of-bound memory accesses.
Draft the design for such a tool and justify your design giving a high level overview over the policy in general and the necessary instrumentation (not more than 100 words). Explain whether (and why) your toolchain instruments the program only at compile time, only at run time, or both. Are there potential drawbacks introduced by your instrumentation? If yes, which and why would they occur? Can you think of optimizations that reduce the impact of potential drawbacks?

Question 33: *This question is worth 20 points.*

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10
☐ 11 ☐ 12 ☐ 13 ☐ 14 ☐ 15 ☐ 16 ☐ 17 ☐ 18 ☐ 19 ☐ 20

Consider the following code snippet:

```c
int maze_game(char* input, unsigned int input_len){
    unsigned int ox, oy, rand_num, i = 0, x = 0, y = 0, score = 0;
    unsigned char *maze = (unsigned char*)malloc(512*512);
    if (maze == NULL) return -1;
    // For simplicity, we omitted the maze initialization which you can assume to be correct.
    // Maze initialization: bytes are initialized as ' ' (pass, multiple), '|',
    // '-', (wall, multiple, appear at random locations on the boundaries of the
    // maze and inside the maze), 'x' (bomb, multiple), '#' (treasure, only one,
    // with coordinates (511, 511))

    // cheat code: win directly!
    if (crc32(input, input_len) == 0xdeadbeef) {
        // `crc32()` returns a 32-bit crc of the passed in char* array
        score = 1024 / i;
        printf("Congratulations, you've won the game! Your score is: %u.\n", score);
        free(maze);
        return 1;
    } else {
        while(i < input_len) {
            ox = x; oy = y;
            switch (input[i]) {
                case 'w': y--; break;
                case 's': y++; break;
                case 'a': x--; break;
                case 'd': x++; break;
                default: i++; continue;
            }
            switch (maze[y*512+x]) {
                case 'x':
                    srand((unsigned)time(NULL));
                    rand_num = rand();
                    if (rand_num < 10) { // the bombs explode with small probability
                        printf("The bomb explodes! Game over :(");
                    return 0;
                    }
                    break;
                case '|':
                case '-':
                    x = ox; y = oy;
                    break;
                case '#':
                    score = 1024 / i;
                    printf("Congrats, you found the treasure! Your score is: %u.\n", score);
                    char treasure_str[101] = {0};
                    FILE* treasure_file = fopen("treasure", 'r');
                    fgets(treasure_str, 100, treasure_file);
                    printf("Your treasure is: %s\n", treasure_str);
                    fclose(treasure_file);
                    free(maze);
                    return 1;
            }
            i++;
        }
    }
    free(maze);
    return 0;
}
```

(a) Provide the line numbers for four lines introducing bugs and state their bug type [stack overflow, heap overflow, integer overflow, use after free, double free, bad free, memory leak, format-string bug, division by zero, `NULL` pointer dereference, unchecked system call return value]. For memory leaks, also list the maximum line number before which the missing `free()` can be inserted; for other bug types, also list the line number of where the bug would manifest (e.g., crash because of `NULL`-pointer dereference).

(b) Which bug(s) is/are hard for symbolic execution to find? List the line number(s) and justify.

(c) Which bug(s) is/are hard for fuzzing to find (the fuzzer mutates the content of ‘input‘)? List the line number(s) and justify.

(d) When fuzzing this function, for which bug(s) is the code coverage feedback not helpful in finding them? List the line number(s) and justify.

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

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

Describe, step by step, how an attacker could get arbitrary code execution on a binary compiled from the source code below. You are not required to write the full exploit code, but you are required to provide low-level details of how the attack works. Be precise in your description (i.e., what you do and why). Generic explanations will not be given points (e.g., "the attacker will find a libc leak somewhere on the stack" is not precise enough, you need to specify where and how).
The binary is compiled with the following mitigations:

- RELRO: full
- ASLR: enabled
- PIE: not enabled
- FORTIFY: not enabled
- CANARIES: enabled

Important notes:

- You may assume there is a win() function in the binary that spawns a shell, but if you use this you are awarded only 50% of the points for this question. For full points, use conventional exploitation methods such as ret2libc.
- You know the exact version of the libc used, and know the offsets between symbols inside said libc.
- The binary is very small, and does not contain almost any useful gadget for ROP chains. The only one you found is a "pop rdi; ret".

```c
void win() {
    system("/bin/sh"); // only 50% of the points if you use this!
}

int ask_question() {
    char buf[16];
    gets(buf);
    return 2;
}

char buf[100];
int main() {
    char* p_buf = buf;
    char not_really[42];

    puts("Hello, What's your name?")
    read(1, not_really, 200);
    printf("Hi %s, here's your cookie: %s", not_really, p_buf);
    printf("So %s, do you like this exam?", not_really);

    ask_question();

    printf("Wow, %s, no need to be salty...", not_really);

    return 0;
}
```
