Exercises - Week 7
Software Security

1. What are the three properties that make a mitigation?

Effective against an attack: it effectively should be a defense. The approach should prevent attacks.
Efficiency: The approach should not add a high computation or memory burden.
Compatibility: Low effort to be deployed: no need to change software or hardware, just add a flag, …

2. Are each of the following approaches a mitigation mechanism? Justify.

(a) **Inexecutable stack**
(b) **Dynamic library linking: Allowing a program to load an external library**
(c) **Sandboxing: Running the process in a isolated space**
(d) **Compiling with different optimization flags**

a. Yes. Preventing code execution in the stack is fast and efficient and it is used in production. Furthermore, it makes exploiting stack-based buffer overflow harder.
b. No. Dynamic library linking doesn't help with exploit prevention. In fact it's a very attractive target for attackers to corrupt a library to exploit the programs which use this library, or an approach to run an arbitrary code in exploited programs which have mechanism to prevent data/stack execution.
c. Yes. Sandboxing prevents a process process from accessing system resources or corrupting other processes.
d. No. Compiling several times cannot ensure that errors are found and/or eliminated. It also does not help isolating. It can even be counterproductive since different compilations may provide a larger surface of attack for exploitation.

3. Symbolic execution and dynamic analysis (fuzzing) are two approaches to automatically find bugs. Symbolic execution provides a full path coverage while fuzzing gives partial coverage. Fuzzing may hit a coverage wall and cannot find samples which lead to new coverage. So why is fuzzing more popular in practice? Is there a way to leverage symbolic execution to get better coverage in fuzzing?

Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. It is based on converting the program into logical equations that cover many possible executions by abstracting the value of the variables. Nice tutorial here: https://www.cs.umd.edu/~mwh/se-tutorial/symbolic-exec.pdf.
Symbolic execution is computation heavy and it cannot scale due to path explosion. Each fork in the control flow doubles the number of paths, resulting in a blow up of the symbolic constraints or the number of evaluated paths. Symbolic execution struggles to scale past programs with few thousands line of code, while production programs can easily reach millions of lines of codes.

Fuzzing is an automated software testing technique based on randomly generating inputs in order to trigger bugs in the code. Fuzzing is a probabilistic process that can leverage feedback from prior executions. Also, fuzzing is “jump started” with seed inputs that conform to valid inputs that already exercise complex functionality.

Even though symbolic execution is complete, it is very expensive and cannot be used when the code is large. Thus, Fuzzing is used more often in practice (e.g., it is used by Google to deal with the huge complexity of their products that have millions of lines of code: https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html).

Whitebox fuzzing allows a fuzzer to check the code of the program, hence it can use the semantic information to generate input through symbolic constraint solving for an unseen path to get new coverage when it is stuck. This allows a fuzzer to leverage symbolic execution to produce new seed inputs for interesting cases and feed those back into the fuzzing process.

4. Branch coverage is a metric to measure how much of the code was executed. Compared to statement coverage which measures if a statement is executed, branch coverage measures if an edge in the control-flow graph is executed. For each conditional jump, branch coverage measures the outgoing edges that are taken (e.g., for an if condition, branch coverage captures if the <u>if</u> or the <u>else</u> branch was executed). Note that branch coverage is stateless: this means that each branch only remembers if it has been executed or not.

(a) Branch coverage is incomplete and does not cover all possible execution paths. Explain why branch coverage cannot cover all paths (hint: branch coverage is stateless, reason about paths, not about individual branches).

Branch coverage cannot cover all paths, because it will try to evaluate every statement to both true and false, but will not try all possible combinations of statements, which represent all possible execution paths (e.g see (b)). In other words, the number of branches is linear to the possible choices, but the number of paths is exponential.

(b) Complete the ? instructions in the example below, of a program that has full branch coverage but incomplete path coverage. Add a memory safety bug (e.g., a buffer overflow or an illegal dereference such as buf[usr1] = usr2) to the program and provide inputs to the program that result in full branch coverage but do not trigger the bug.

```
int example(bool b1, bool b2) {
        int a = 0;
        char c[2];
        ?
        ?
        return c[a];
}

int example(bool b1, bool b2) {
        int a = 0;
        char c[2];
        if (b1) { a += 1; }
        if (!b2) { a += 1; }
        return c[a];
}
```

In this example, branch coverage will test this function by calling both example(true, true) and example(false, false), it will execute all the possible branches. These two examples provide a full branch coverage, but example(true, false) results an undiscovered path which leads to a memory safety bug.
To have a complete path coverage, we should test to call example(true, false) and example(false, true), to be sure to check every possible path.

4. Fuzzing is an efficient automatic testing technique that scales to large code bases. Modern fuzzing mechanisms leverage branch coverage to record which parts of the program have been executed, mapping fuzzing inputs to coverage. Coverage-guided fuzzers add any input that triggers new coverage to the pool of inputs to perform a mutation. Additionally, these fuzzers record any input that crashes or hangs the program.

(a) Assume a new seed covers a new path. Fuzzing will continuously mutate this input to trigger different paths and different data-flow along that path. Why is it necessary to generate alternate data-flows to trigger bugs, i.e., why does it not suffice to only generate new paths? (hint: what is the difference between control-flow and data-flow?)

Testing can only show the presence of bugs, never their absence. The number of possible paths is too high (exponential), so the programmers cannot test the program for every input, let alone manually feeding them to the test. The initial set of inputs provided by the tester is usually small. Generating input manually is costly, and we want to get high coverage in the testing, that is why we use the initial pool as a seed to generate more samples, to have a higher chance of finding bugs.
Executing a path only covers control-flow. However, a bug may only be triggered using specific data-flow. This means that both the control-flow and data-flow must match to trigger a bug: control flow checks whether an instruction executes or not, but the result of the execution depends on the data. Some data may be okay while there are inputs which result in a bug. See the example below.

```
Int example(int a, b){
        if ( a > 0 && b > 0 )
                return  a / (b-50);
        return 0;
}
```
(-1,-1) and (1,1) cover all path flows in the function, but (1, 50) results in divide by zero exception which cannot be detected by control flow alone.

(b) Fuzzing frequently hits a so-called "*coverage wall*" where it no longer makes progress (i.e., random mutations do not trigger new coverage). What could be the reason for this limitation? (hint: what types of conditions are hard to satisfy for randomly generated input)

Programs often have several checks in sequence. If these checks are complex it is unlikely that a fuzzer will randomly generate input that satisfies all these checks. For example:

```
long long input[2]
if (input[0] == 0x12345678) {
        if (input[1] == 0x87654321) {
                bug();
        }
}
```
Here it is unlikely that a fuzzer will generate input that is exactly "1234567887654321". The probability of generating this input is 2^128.

Long answer:
To execute a specific path in the program, the input needs to satisfy all of the conditions in the branches, and this results in a complex symbolic satisfaction problem (finding a set of inputs which satisfy all of conditions in all of branches is difficult). If the number of inputs which can satisfy this problem is small, it is unlikely that a random set of inputs can satisfy the constraints. Security requirements are one of the hard conditions to satisfy. Another example, if a program needs a signed input, probability of feeding a correct random input with a correct random signature would be negligible. (If you can randomly guess a signature the signature algorithm not secure!!!!).

(c) Fuzzing struggles to find crashes in libraries. What could be the reason for the lack of deep coverage when fuzzing the set of exported library functions? (hint: think about a file I/O library that offers open/read/write/close functions; what happens if you only fuzz the read function without prior calls to open?)

A library can consist of several features and functions, all of which might not be used in the program that is using it. Furthermore, it's infeasible to determine whether a branch is unreachable from the main code or the fuzzer didn't find any input which invokes that function. If only some of the functions (present in the program) are fuzzed, there might be dependencies with other functions that are missed.
Furthermore, libraries often have complex dependencies and state between calls. Without building the necessary state, execution will bail early without executing the interesting parts of the library.

5. Sanitization makes bug detection more likely by enforcing certain policies. Commonly used sanitizers enforce memory safety and detect undefined behavior.

(a) Is sanitization instrumentation helpful to find all types of bugs? Under what circumstances will sanitization be counterproductive?

We need to be able to detect and determine a bad behavior when using a sanitizer. A sanitizer can detect undefined behavior or out of band address, but it cannot determine a wrong behavior if it is based on the program specifications.
Moreover, sanitization has a high cost and each check that we add impacts the performance. Hence, running a time-consuming sanitization in production can be counterproductive.

Sanitization can test for bugs that violate a codifiable policy. If the bug is not encodable as a policy then no sanitizer can be built to find it and sanitization will there not find it. An example is the libssh bug that allowed an adversary to send a message "I'm successfully logged in" despite not having sent the password. This was an error in the SSH state machine that would not be testable through a sanitizer.
https://www.zdnet.com/article/security-flaw-in-libssh-leaves-thousands-of-servers-at-risk-of-hijacking/

(**b) Why is address sanitizer needed to detect memory corruption? Explain why and how buffer overflows can be missed without address sanitizer.**

Every memory corruption, such as buffer overflow, doesn't necessarily lead to a crash. If the overflow address is mapped in the address space of the program, the OS allows the program to change the content and continue the work. Moreover, this corrupted memory may or may not lead to a crash/wrong output in the future. Hence, sometimes overflows don't impact the flow of the program. These inputs are not problematic, but knowing the reason behind this overflow and fixing it can help us to prevent possible bugs based on this overflow without finding an input that triggers the bug.

(c) Address sanitizer detects memory corruption by detecting writes to red-zones (8 byte areas directly adjacent to allocated memory with static data). Why is address sanitizer not a mitigation?

If an attacker knows about the red-zone, she can move the pointer past the red zone and thereby adjust the exploit to work in the presence of address sanitizer. Address sanitizer only detects writes to the red zone but does not protect or check the pointer itself. The red-zone can detect an accidental overflow or and unsophisticated attack, but it can be bypassed (i.e., it is not effective against an attack). Furthermore, adding 8 byte to each variable would have a very high impact on the memory usage which makes this method impractical for production (i.e., it is not efficient).

6. Several mitigations exist to make exploitation harder. Mitigations must adhere to strict performance criteria as they are always enabled.

(a) ASLR shuffles the address space for each execution. Why can the address space not easily be reshuffled during execution (e.g., after each system call)? Why would it be useful to reshuffle after each system call (think how many stages an attack will have)?

The program needs to reread and rewrite the whole program address space to allow function pointer reshuffling. All pointers in the address space must be adjusted to the new layout. This requires the system to keep track of all pointers. This is very costly!

It would be useful (if it was efficient) because the adversary needs to complete the exploit inside one call because, after each call, the system shuffles the memory space and invalidates the previous attacker pointers.

(b) Data Execution Prevention stops code injection attacks. Initial implementations of data execution prevention leveraged segmentation registers and expensive checks to test if a memory region was executable or not. Modern implementations use a page-based mechanism that leverages a bit in the page table to encode execute permissions. Discuss the key advantage and disadvantage of a page-based solution

Disadvantage
Granularity of protection is page based which means that if a single memory page contains code and data then it must have a superset of both privileges. If the data is writable then the whole page is writable which could lead to code injection attacks. Modern DEP implementations make sure that pages are either executable or writable though.
Advantage
Very low overhead as the check is executed implicitly by the memory management unit during address translation and the lookup is cached in the translation lookaside buffer, allowing a very fast check in practice.

(c) Stack canaries protect against buffer overflows on the stack and are prone to information leaks. How could an attacker bypass stack canaries in an exploit?

An information leak allows an attacker to disclose memory content. The attack uses an information leak in the first step of the attack to disclose the content of the canary and returns this content to the attacker. The attacker then adjusts her attack to overwrite the canary with the correct value. Stack canaries do not enforce integrity at all times but only check integrity when the function returns, this allows the attacker to change the content at any time inside the attack before the function returns.

The questions below are on topics not explained in the class, and thus not part of this course material. However, since they appeared in the first published version of the exercises we include the answers for completeness.

(d) CFI restricts calls to a limited set of targets. The effectiveness of the mitigation depends on the precision of the analysis. To get really precise target sets, an analysis must collect information across all libraries and the full program and synchronize the

analysis (otherwise, targets would be missed). Why is it challenging to merge target sets across different programs on the system? Discuss the security/precision versus compatibility tradeoff.

To prohibit false positives where CFI would terminate a program for a legitimate call, the call target sets need to be synchronized across all libraries. As libraries may be used in different programs, the sets cannot be statically encoded in the libraries itself but must be encoded on a per-program basis, based on the libraries they include. This requires either an over-approximation where a vast number of targets are allowed (e.g., Microsoft's control-flow guard allows any function as valid targets (does not allow jump to the middle of a function) whereas Google's LLVM-CFI requires Link-Time Optimization to merge the target sets for an application). This is an inherent trade-off between precision and compatibility.

(e) CFI focuses on the forward edge (indirect calls) and not the backward edge (function returns). Protecting the backward edge is challenging because functions may be called from many locations, resulting in large amounts of backward targets for each return site. What are alternatives to CFI that increases the precision on the backward edge? (hint: CFI is stateless, the stack needs state, so how can you keep the context about the stack?)

Having static analysis for backward edges is infeasible. For example, functions like open and printf cover a large portion of program codes and they are interesting for the attacker. Hence, we keep state for CFI and use a dynamic backward edge tracking. Every time that we call a function, we add the function to a backward edge list, which is outside of the stack memory). Whenever the program wants to return, it matches the return address with the memory space of the caller function. An attacker needs to change both the return address and the backward list to inject the code and since the list is not inside the stack and not accessible with buffer overflow, then exploiting it is significantly harder. This policy is also called a shadow stack which captures the dynamic nature of call/returns: a shadow stack keeps track of all invocation frames but without all the data. This slim stack can be protected and checked efficiently.