Final Review Session

1 Overview

Part 1 of the course covered many cryptographic primitives. You should know them! But Part 2 of the course (covering platform security, software security, and human/end-user security) provided a survey on many systems, attacks, and general security practices that are important to understand.

- Lec 17 - software trust
  - Imports, updates, secure boot
- Lec 18 - hardware attacks
  - Side channel, row hammer Physical attacks
- Lec 19 - iOS security
  - app sec, secure boot, BOOT ROM, secure effaceable storage, Secure Enclave
- Lec 20 - Software Security
  - Bugs: buffer overflow, use after free, encode (SQL), XXS, concurrency bug
- Lec 21 - Privilege Seperation
  - examples: logging, keys, codec, NTP, OpenSSH
- Lec 22 - Bug finding
  - Fuzzing, symbolic execution
- Lec 23 - Runtime Defenses
  - Stack canary, fat pointer, ASLR, non-exe stack
- Lec 24 - Schnorr
  - Discrete Log Problem review, Schnorr, Zero-Knowledge
- Lec 25 - Diff. Privacy
  - Laplace Mechanism for noise

2 Review Problems

Problem 1: Bug Finding
Let's assume we are using a symbolic execution system in order to find any bugs in the following small piece of code:
We notice that when a = 0, b = 2, and c = 1, our code crashes.

```
int x=0, y=0, z=0;
if(a) {
    x = -2;
}
if (b < 5) {
    if (!a && c) { y = 1; }
    z = 2;
}
assert(x + y + z != 3);
```

Draw an execution tree for symbolic execution on the above code. How many different symbolic execution paths are there in the above code?

Solution There are 6 paths. The root could be the possible values for symbol assigned to a, branching out 2 times for true or false. Then there is a branch on symbol assigned to b, branching out at < 5 and ≥ 5. After b $\geq$5, there is a leaf in our execution tree, the code asserts. If b < 5, then there is another branch based on the value of 6. This totals to 6 execution paths, exactly one leads to failed assertion.

Part b. Now, let's say we are using a fuzzer to find the same bug. Let's assume that the code coverage mechanism and corpus for said Fuzzer does not significantly impact runtime of the fuzzing process, and effectively the Fuzzer picks Fuzzed inputs uniformly at random (which is not true in practice). If inputs `a,b,c` are all obtained from `ConsumeUInt(2)`), what is the expected number of inputs before our Fuzzer finds the bug?

Solution Consume Uint2 consumes a 2 byte integer, (`uint2`). The max value is $2^{16} - 1$, the minimum value is 0.
To trigger the assertion, $p(a = 0) = 1/2^{16}$
To trigger the assertion, $p(b < 5) = 5/2^{16}$
To trigger the assertion, $p(c \neq 0) = (2^{16} - 1)/2^{16}$

So, if all are independent, then on expectation it would take: $2^{16} * 2^{16}/5 * 2^{16}/(2^{16} - 1)$.
Fortunately, in real life, the Fuzzer's code coverage detection would likely eliminate the need to have each occur randomly at the same time (realizing that for example when $a = 0$, there is more code run, so it will try $a = 0$ more often when randomly setting b and c).

Part c. What are the is a pro of using a Fuzzer over Symbolic Execution?

Solution Faster

Part d. What is a pro of using Symbolic Execution over a Fuzzer?

Solution Exhaustive so easier to catch rare cases systematically.

Problem 2: Schnorr Solution
Recall the Schnorr signature game from lecture, say the Prover knows c=1 before the protocol begins. Describe how, without knowing x, the Prover could still pass the game, tricking the Verifier.

Solution If the Prover knows c=1, then it could set the value of z to anything it wanted, and trickily set $R = X^{-1} \cdot g^z$, This way, $R \cdot X = X^{-1} \cdot g^z \cdot X = g^z$. So the verifier will always accept: $R \cdot X^c = g^z$.
Similarily if the prover knew c=0, it could simply always return $z = r$, and never need to know x.

Say the verifier could run the protocol twice, once with c=0 and once with c=1, how describe how the verifier could obtain the value of x.

Solution If the verifier runs the protocol twice, once with c = 0 and once with c = 1, then it can take both z values returned and subtract them to find x.
$z_{c=0} = r$
$z_{c=1} = r + x$

$z_{c=1} - z_{c=0} = x$

This only works because r has not changed (as it normally would when re-running the entire protocol).

Consider a modified version of Schnorr's signature in which the signing nonce r is computed as $r \leftarrow H(m)$, where $H : 0, 1^* \rightarrow Zq$ is a hash function, m is the message to be signed, and q is the order of the group used for the signature scheme. Is this deterministic version of Schnorr's signature scheme is secure?

Solution This would break for the reason above, because when verifying the same message multiple times, we would have a constant $r = H(m)$, which we could then subtract out to find the secret key x.

Problem 3: Differential Privacy
Your Laboratory, Happiness Inc, conducts a top secret survey, which polls the age and average reported happiness of 8, very private individuals. You want to employ differential privacy techniques while releasing some statistics about your data.

age	happiness
14	.1
10	.5
20	.8
30	.7
40	.1
61	.1
43	.2
12	.1

Assume when generating D' , a data set with one row changed, to determine GS_f we pick a value within the minimum and maximum value of above data range and maintain 8 total records.

Part a. Let's say we want to apply differential privacy with ϵ to report the mean of our `age` data. What Laplace noise would we want to use? What about for the mean of our *happiness* data?

Solution Generate Laplace noise with mean 0, with $b = \frac{GS_f}{\epsilon}$. Then, return the true stastic plus the noise value.
Age:

$$GS_f = (61 - 10)/8 = 51/8$$

Happiness:

$$GS_f = (.8 - .1)/8 = .7/8$$

Part b. Let's say we want to apply differential privacy with ϵ to report the maximum of our age data. What Laplace noise would we want to use?

Solution Generate Laplace noise with mean 0, with $b = \frac{GS_f}{\epsilon}$. Then, return the true stastic plus the noise value.
Age:

$$GS_f = 61 - 43 = 18$$

Happiness:

$$GS_f = (.8 - .7) = .1$$

Part c. Which statistic requires more noise to report?

Solution The Maximum...

Problem 4: Isolation
Let's say we are implementing an isolated system using Time Multiplexing, for example the video game system discussed in class. Does the system provide integrity, non-leakage and non-interference? What assumptions do you need to make in order to acheive each of these isolation goals?

Solution Integrity as long as the loading system does not have bugs and is not exposed to the running video game. This is because one game runs, has all state on hardware, then offloads the result/status of machine when it is done. As long as a running game can not tamper with save states, our integrity plan is good (we can not change the state of the victim non-running game by running A).
We have non-leakage as long as the storing system is not readable by the adversarial running game for similar reasons.
We have non-interference only if the adversarial game has no connection to the outside world or timing information. With timing information it may be able to detect when it is not run as often (because the victim game was running for that time). Alternatively, if the system runs each game at a fixed interval regardless if there are multiple processes present.

Problem 5: iOS
Match the following properties to the component. A property may apply to more than one component.

Components	**Properties**
Boot ROM	Effacable
Secure Storage	Non-Writeable
Secure Enclave	Contains Guess Counter
Bootloader	Contains a Verification Key
	Contains an Encryption Key
	Enforces PIN Delay
	Part of Secure Boot
	Part of User Data Protection

Solution Boot Rom, is non-writeable, contains a verification key (for the bootloader) and is a part of the secure boot process.

The secure storage contains an encryption key (shared secret communication channel with enclave) and the root user AES encryption key. It is effacable and contains the guess counter, it is a part of user data protection story.
The secure enclave enforces the pin delay, contains an encryption key for the secret communication channel with secure storage, and is a part of the user data protection story.
The bootloader contains a verification key for the kernel and is a part of the secure boot process.

Problem 6: Privilege Separation
Let's say that the NTP (Network Time Protocol), instead has the Time Service and Network service running in one process. What could potentially go wrong with this plan?

Solution Because the Time Service requires root privledges, if the network service is compromised (lots of code in network stack), then the adversary could have root privledges to the system, which is very bad!

Recall the Logging Privledge Seperation plan from lecture. Let's say our application server has direct access to our Logging database, what could potentially go wrong with this plan?

Solution If the application is compromised, then the adversary could have the power to trigger a deletion of all logs, covering their tracks. With these components separated, the compromised application does not have access to API to trigger deletions,etc.

Problem 7: Runtime Defenses
Assume that p is a fat pointer. Which of these scenarios would fail/succeed?
Scenario 1:

```
p = malloc(4);
*p = 1;
```

Scenario 2:

```
p = malloc(4);
q = p;          // copy fat ptr to q.
q = q + 8;      // add 8 to q.curr.
*q = 1;
```

Scenario 3:

```
p = malloc(4);
q = p;          // copy fat ptr to q.
q = q + 2;      // add 2 to q.curr.
*q = 1;
```

Scenario 4:

```
p = malloc(4);
q = p;          // copy fat ptr to q.
q = q - 1;      // subtract 1 from q.curr.
*q = 1;
```

Solution A fat pointer adds a base address and a limit address which represent the lower and upper bound of memory where accesses are okay.
Scenario 1 is a normal memory dereference operation at the pointer returned by malloc, this is OK.

Scenario 2 adds 8 to the pointer address and attempts a memory dereference operation. However + 8 will be out of the 4 bytes allocated by malloc and greater than the limit values of the fat pointer. This will FAIL.

Scenario 3 adds 2 to the pointer address and attempts a memory dereference operation. However + 2 will still be within the base limit range. This is OK.

Scenario 3 subtracts 1 from the pointer address and attempts a memory dereference operation. Subtracting 1 from the malloced address is less than the base pointer. This will FAIL.

Name 2 downsides to using fat pointers.

Solution Takes up 3x amount of space to store pointer.
Slow to check within bounds on each pointer deference operation!
Will not enforce rules on structs,etc, for unintended memory accesses within ONE allocated section.