April 5th Lecture: Side Channels
Lecturer: David Wagner
Discussion Lead: Pranav
Scribe: Neil

BREACH: Reviving the Crime Attack

- Compression correlated to content of data and length
- Idea is to look for substrings that appears multiple times in input, and replace these long substrings with a short substring (creating a mapping)
- Setup: Attacker is eavesdropper, using HTTPS (and thus sees only encrypted packets)
- Goes after CSRF token (prevents CSRF attacks)
- Assumption: Victim will also visit attacker's website, and attacker can get victim to visit link of attacker's choosing
- On many bank websites, the web page includes bank info, CSRF token, and some information from the request
- Data from the response (which is encrypted and compressed), contains something from the url, some data that is secret, and some from the attacker
- Attacker guesses first nibble of the secret data (canary value), so that it will be echoed back
- For example, if it starts with canary=3 then it compresses to something shorter if the attacker's guess is incorrect, otherwise the length will be the same if attacker is correct
- Attacker repeats for each nibble of the canary value (uses leak of length to check whether guess is correct)

How to Prevent this Attack?

- Do not compress (the deployed solution)
- Have a universal dictionary (for substring mappings) that is fixed and public for everyone
- Mask secrets
- Compress only the non secret parts (might require assistance from the web server)
- Add a small chunk of characters at the beginning to bound the number of queries the attacker can make

Power Analysis Side Channel Attack:

- Smart card (credit card chip) contains some crypto keys, circuitry on the card, and it'll ask the chip to sign to authenticate
- Power is supplied by the card reader, if card is inserted to malicious reader (one that can measure the power draw), the reader can measure how much power is being used, and then use that information to learn some secrets

Pseudocode:

D - secret

Y = 1
For i = lg(n)..0:

```
 Y = Y^2 mod n // high power consumption (squaring more efficient than multiplying)
 If ith bit of D is 1:
        Y = Y*X mod n // high power
 Do nothing // lower power
Output Y
```
(Squared multiply)

Analysis:

- If branch taken then 2 multiplications occur so more power is drawn
- Can identify when i is being incremented, when squared, when multiplied, etc.

Root Password Example:

```
Func compare(x, y, n):
   For i=0,1,..,n-1:
      If x[i] != y[i]:
         Return False
   Return True
```

Time it takes is proportional to how closely equal the two strings are, and thus attacker can time how long it takes (longer is more correct), and repeat for each character (some noise from network delay, cannot measure exactly how long it takes for server to do this comparison).

```
compare(x, y, n):
   Flag = True
   For i=0,1,..,n-1:
      If x[i] != y[i]:
         Flag = False
  Return Flag
```

To fix, make the code take the same amount of time (constant time). One idea is to compare the SHA hashes of the two strings, which will make the correlation not present.

Randomization/Noise

- Noise less effective than one might expect

Example:

Y = measured voltage
If 3V eel: Y ~ N(3,1)
If 5V eel: Y ~ N(5,1)

Compare against a threshold
If Y > 4 guess 5V eeel otherwise guess 3V eel

Y_i the ith measurement
If 3V eel: Y ~ N(3,10^2)
If 5V eel: Y ~ N(5, 10^2)

Huge overlap, so can't really use threshold much higher error
Z = average of every Y_i
If 3V eel: Z ~ N(3, (10^2) / n)
n ~ constant times variance of the noise

Randomization can allow repeated measurements so attacker can break with enough measurements

Speculative Execution Attacks:
SPECTRE

- Modern processors have more than 10 pipeline stages
- Failure to predict next instruction can be slow because of large pipeline
- There exists some cache shared by all processes

```
Void victim_function_v01(size_t x)
{
        Int Temp;
       If (x < array1_size) {
            Temp = a1[x];
            Value = a2[Temp * 4096];
       }
}
```

Can trick to always take branch, even when it shouldn't have

Setting up attack:

- Attacker flushes the cache
- Array1_size and a2 uncached
- Secret cached/uncached
- Attacker trains branch predictor to predict x < array1_size
- Read for array1_size, and speculatively execute
- Read address a1+x; x out of bounds (address at 0x04)
- Secret value resides in Temp (address at 0x04)
- Read address a2 + temp*4096 (address at a2 + 0x04*4096)
- Attacker finally measures read accesses for a2[y*4096], for y = 0,1,2,3,4
- Attacker can probe each cache line, and figure out which one has changed

Defenses:

- Modify the program: Make sure to never speculatively do loads and stores after branches by using memory fences (which forces all the speculation to be resolved before executing instructions)
- Isolate cache lines or flush every time you do a context switch
    - TEEs give each enclave its own set of cache lines