CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 18th, 2011

High level security design flaws

Last time: Run-time taint tracking

Where is the tag-map stored? In memory, which means, just like any other thing in memory, it could be tainted as well.

Attack: If you can write to the tag-map arbitrarily and mark tainted memory as untainted, then you can compromise the security of the system.

Every time there's a memory write, such as:

```
tagmap[p] = tainted;
*p = value;
```

We will fail the program if `p` points within the tag map.

Solution: Make it such that accessing `tagmap[tag_map_address]` will cause a runtime error.

This way, if an attacker does a write at `tag_map_address` using something like `*tag_map_address = 5;` that write would never be reached because the precedent `tagmap[tag_map_address] = tainted;` instruction would have caused a runtime error.

How? We can make the section of the tag-map that tracks the taint of the tag-map itself not have any pages allocated for it, such that when `tagmap[tag_map_address]` is dereferenced an error occurs and the program exits.

Runtime overhead:

- In real server applications: $\sim 3-5\%$
- Compute benchmarks: $\sim 50\%$

Overview of high level security design flaws

An interesting class of attacks involves not the main communication channel, but side channels, such as timing (how long an operation took):

- Side-channels
  - It can be timing, power, light, heat, cache state, etc.
- Covert-channels
- Reaction attacks
- Remanence

Smart Cards

A **smart card** is a card with a CPU and various cryptographic abilities. (*Funny fact*: most of the times these things run Java code)

Consider the **RSA private key**, a sequence of bits computed with the modular exponentiation algorithm. When you decrypt/sign with RSA, the algorithm looks something like this:

```
for(int i = 0; i < num_bits; i++)
{
```

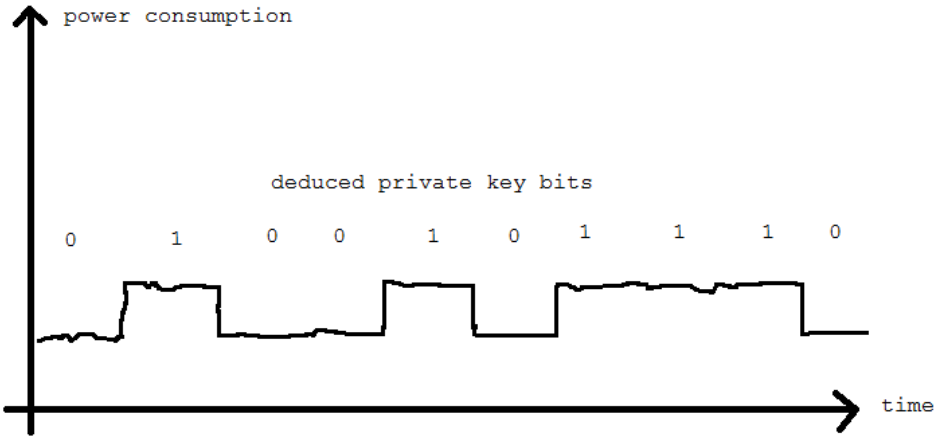
```
        if(secret[i] == 0)
                do A
        else
                do B
}
```

This algorithm runs on the smart card. Note that the **card reader provides power** to the card.

Reality: There could be **evil merchants** that try to steal your private key from your smart card. In the past, there has also been fraud where people broke into the merchants' stores and replaced the card reader boxes with evil ones that stole the card information.

- Obviously then, your smart card has to do its best not to reveal your secret key.

Attack: Since the merchant box provides card with power, the **merchant box will know how much power the card uses** at any time t. It turns out that since A and B take very different amounts of power, the box can **deduce the private key** from the power usage of the card.



RFID chips

The RFID reader generates such a strong EM field that the RFID chip harnesses the energy and uses it to perform some computation and send an ID number (or some other information).

RSA timing attacks

If you're not careful when implementing an RSA algorithm (programmers try to get RSA to go fast, so they do lots of optimizations), you can end up with serious security holes in the implementation.

Some **optimizations can depend on your key** or **depend on the data that is processing**.

In one attack, an SSL server was set up and a bunch of carefully crafted requests were sent whose response time was measured. The time differences between requests were around microseconds, with network latency noise. However, the key was successfully extracted from the timing info.

Timing attack

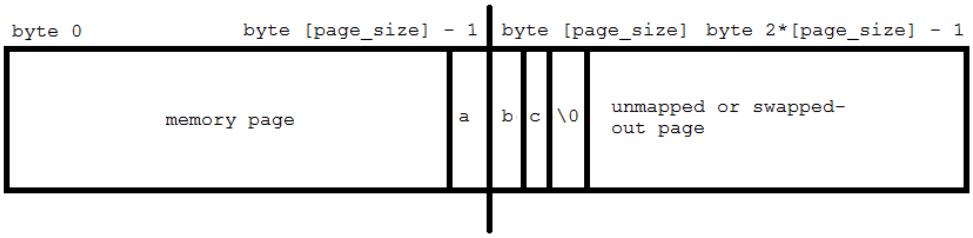
Consider the following login function, which is vulnerable to a timing attack. We will show that an attacker can successfully deduce the password by doing consecutive timed calls to `login`.

```
bool login(char * username, char * password)
{
	char * realpassword = lookup_password(username);

	//	Ignore the length bug here, and assume this was done right.
	for(int i = 0; i < strlen(realpassword); i++)
		if(password[i] != realpassword[i])
			return false;

	return true;
}
```

To guess the first character, the attacker can create a two or more characters password **at the end of a virtual page** such that the first character is the last byte in the page and the other ones go into the next page, which the attacker will ensure is swapped out (or even unallocated). He will then pass this password to `login`.



Important note: Really, you can think of the `login` function as having **three return values**: yes, no, page fault/swap in. You can use the page fault/swap in errors to build a **reaction attack.**

Important realization: If the first character of the password is correct, the `login` function will return a page fault/swap in.

- **Why?** Well, if the first character of the password is indeed the correct one, then the login function will attempt to match the next one which is in the next page. The access to the next page will generate a page fault/swap-in, which the attacker can confirm by timing how long it took the function to finish executing. With a page fault/swap-in, the function will take considerably more time to execute.

The **WEP (Wired Equivalent Privacy)** standard had a similar reaction attack where the encryption key could be extracted.

Cache timing attacks

The CPU has a cache and will swap control from one process to another process. Consider our previous code

```
for(int i = 0; i < num_bits; i++)
{
	if(secret[i] == 0)
		do A
	else
		do B
}
```

What if “do A” and “do B” access different memory locations? An attacker could tell which one is being executed by timing the cache in a clever manner.

The cache entries for A or B will push out some of the attacker’s entries from the cache. Suppose A pushes A’ out and B pushes B’ out.

If the attacker can preempt the `for` loop at every iteration (not hard to do), then the attacker can see how long accessing its A' and B' entries takes and, depending on the time it took, he can tell which function was executed (A or B).

Scheduling talk: Ideally a process should not be able to steal CPU time from another process. However, a lot of schedulers give I/O bound processes higher preference. The scheduler determines whether a process is I/O bound if its time quantum didn't expire.

- When a process does I/O, that process will stop early and the scheduler will notice that the process did not fully

Therefore, a process can get higher priority by doing some I/O at the end of its time quantum.

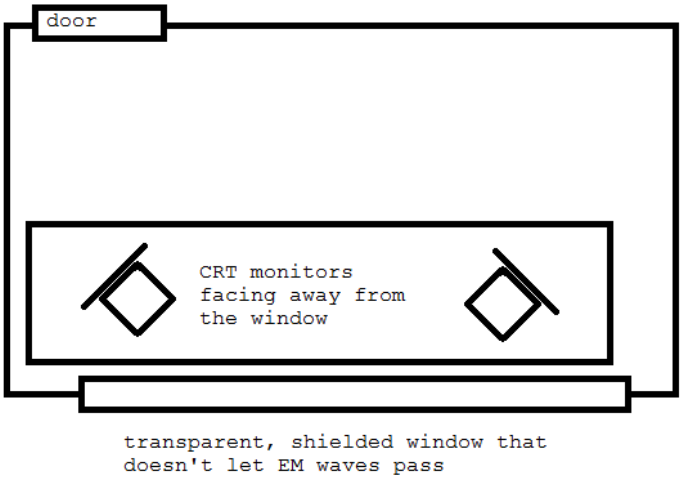
AES algorithm cache attacks

AES has a gigantic table to make computations go faster. The accesses to the table depend on the key. You can play some games to get parts of the table pushed out of cache.

The ultimate coolest side channel attack

Imagine a research lab, where the researchers are using CRT monitors, turned away from the windows, such that no one can spy on their research.

Using a high speed camera, you can see a pattern of changing brightness projected on the wall which you can correlate with each pixel color changing on the screen.



It turns out that with enough precision, you can reconstruct all the images displayed by the CRT monitors.

Data remanence

If the HDD is encrypted with a tool like BitLocker, then the encryption key has to be stored somewhere in memory unencrypted (no TPMs involved).

If the attacker yanks out the batter, then the laptop instantaneously dies and the RAM will clear. But will it? As it turns out, no, it won't. Not until a few minutes later.

Attack: After shutting your machine down, an attacker can boot your laptop with a USB device that scans the memory looking for the AES key, assuming they can do it under a minute or two.

- **Bonus:** If you cool the RAM down you can make it hold its memory for longer

Possible fix: You can try and store the AES key in a CPU register that only the OS can modify.