Disposition 6: Threats and Pitfalls

Mathias Ravn Tversted

January 6, 2020

Table of contents

What attackers want: STRIDE

STRIDE is a way to categorise attacks according to what the attacker is trying to achieve

- **S**poofing Identity
- **T**ampering data without being detected
- **R**epudiation. Attackers are able to deny their actions
- **D**enial of service
- **E**levation of privilege

The means used: X.800

One can categorise attacks according to the *means* used. The X.800 standard is published by the CCITT organisation. **Passive attacks**: **Eavesdropping** attacks where the attacker intercepts information. **Traffic analysis**, where the attacker looks at who is sending what to whom and how much is sent.
Active attacks: These are **Replay attacks** where the attacker replays old messages, **blocking** attacks, where the attacker stops messages from reaching their destination. Also **modification** attacks, where the attacker changes information that is being sent.

Where are we attacked, and by whom: EINOO

First we can look at who attacks us

- **E**xternal attackers, who are not legal users of our system
- **I**nsiders, who are registered and legal users of the system

Then *where* the attacks come from.

- **N**etwork attacks. The adversary can listen and modify network traffic. This is described in part in X.800
- **O**ffline attacks. The adversary gets access to disks or other hardware, perhaps even stealing payroll information of a hard disk (like what happened to Facebook). These are harder to

Where did we make the mistake: TPM

We can also categorise attacks according to what allowed them to happen

- **T**hreat model: The attacker is possible because the threat model was not complete
- **P**olicy: The attack is possible because the specification is wrong or was too complicated to be implemented etc
- **M**echanism: The attacker is possible because the security mechanisms were circumvented or broken

Illegal input attacks

Using illegal inputs, it is sometimes possible to attack a system, such as with SQL injection or buffer overflow attacks. Here are some (covered) examples of illegal input attacks.

- Overflow attacks
- Unicode Exploit
- XSS
- Heartbleed

Overflow Attacks

An overflow attack hapens when, for example, an adversary gives an input that is longer (or perhaps shorter) than expected. A good example of these kinds of errors happens to programs written in C, because it does not do checking on arrays and so on. A classic example is the Heartbleed bug, and the following example

```
1 void foo(char* input) {
2       char buf[3];
3       strcpy(buf, input)
4 }
```

Overflow Attacks

WHAT KIND OF ATTACK MODEL DO WE USE HERE? RELATE TO TAXONOMY Since strings in C are 0-terminated, it will continue to copy until it sees the 0. This means that it will write out of bounds of the buffer, and perhaps even into the previous stack frame. This could impact things such as the base pointer and stack pointers or other variables, which are now under the control of the adversary.

The Unicode Exploit

WHAT KIND OF ATTACK MODEL DO WE USE HERE? RELATE TO TAXONOMY When trying to access directories on a web server, a user may send an URL or some other string that defines what resource a client is trying to gain access to. An example of this going wrong was the IIS (Microsoft internet server, such as for ASP.NET) checking these requests. FRA BOGEN: "At the same time, requests are allowed to contain unicode characters which is a way to encode special characters like ø, å using only the "normal" character set. This allows directory names to contain all kinds of international characters. These characters must be decoded before taking action. Unfortunately, the IIS (or at least a previous

Cross-Site Scripting

WHAT KIND OF ATTACK MODEL DO WE USE HERE? RELATE TO TAXONOMY Cross-Site scripting seeks to exploit poor design of web pages. There are many variations, and this is one particular kind of cross-site scripting. Say an input to a page shows that input to others (think comment section), it may be possible to inject Javascript into that input. This Javascript may make a request to a site that the adversary controls, and so the adversary is now able to control pretty much everything. This means that sensitive information may be sent to the adversary.

Heartbleed

WHAT KIND OF ATTACK MODEL DO WE USE HERE? RELATE TO TAXONOMY The Heartbleed bug was a bug in the open-source OpenSSL implementation of the SSL/TLS protocol. SSL/TLS has a heartbeat feature, where clients regularly confirm that the server is alive. It sends a nonce as well as the length of the nonce. The server copies the input string and echoes it back to the client. Unfortunately, a client could lie about the length of the input string, and when the server then copied the input, it would also copy garbage memory and echo it back to the client. The client could then read the memory of the server.

IBM 4758

The case of the IBM 4758 was a case of the security policy being incorrectly specified. It's a hardware unit with a protected mechanism. The box is accessed through an API. It was possible to extract cryptographic keys from the box, by using the operations made possible by the API in a clever way. For compatibility reasons, the box did both single DES with a 56-bit key, and a two-key triple DES key, where two DES keys are concatenated to form a 112 bit key.

IBM 4758: The Attack

The attack goes as follows WHAT KIND OF ATTACK MODEL DO WE USE HERE? RELATE TO TAXONOMY

- Have box create single length DES key K_0 and compute it using exhaustive search. Have it encrypt some known text. This is possible because it is a 56-bit key
- Ask it to form double length *key encryption key* K_2 with *replicated halves*. This is allowed because K_0 is a single length key so it is not stronger than K_2
- The above step produced $E_{K_2}(K_0)$. We can now find K_2 with exhaustive search as above. Since K_2 is a *key encryption key*, that means it can be

PRG

A *pseudorandom generator* can be used to extend something random, such as random noise from your computer (think random device on UNIX systems). However, this randomness from the entropy of the computer is usually not enough. Therefore this is expanded with a pseudorandomgenerator. A PRG is an algorithm G which takes an input state s_i, and outputs a random *looking* string r_i, and the next state s_{i+1} like so: $(r_i, s_{i+1}) = G(s_i)$. This PRG is initialised with s_0, known as the seed. This should be kept secret, otherwise it is possible to predict the sequence of outputs.

DUAL EC DBRG

Using two points P, Q on an elliptic curve, you can make a PRG. It looks like the following

- Take as input s_i (between 1 and n)
- Compute next state $s_{i+1} = p^{s_i} \mod n$
- Compute output $r_i = Q^{s_i} \mod n$

If P, Q are random, it is hard to break to break the PRG. However, if they are hardwired (like a NIST standard), then it is possible to break it

- Adversary chooses random Q and x. Then computes $P = Q^x \mod n$. Adversary gives user P, Q to the user and keeps x
- Adversary observes r_i from PRG, advresary computes $s_{i+1} = (r_i)^x \mod n$, and can compute all future outputs because $s_{i+1} = (Q^x$

SPECTRE

The SPECTRE attack makes use of *speculative execution*, in which the CPU attempts to predict the result of a fetch from memory. Instead of waiting for memory to be fetched, it attempts to predict what the outcome will be, and begins execution of the instructions it thinks it will run. If however, the CPU guesses wrong, it will roll back to the previous state, however this is not always so.
Consider a program like so

```
1 if (x < array1_size) {
2 y = array2[array1[x] * 4096]
3 }
```

SPECTRE

- Attacker controls x, which may be an input to the program
- CPU believes it will evaluate to true if the adversary makes it so many times
- x and array1 are in cache, but array2 and array1_size are not (array1 size is a variable)
- Adversary can control cache by using some variables more than others

Attacker can now execute with a value of x that is too large, and array1[x] can now be almost whatever they want of the process memory. CPU will now fetch array1_size and execute the body. $\text{array1}[x] \cdot 4096$ can be evaluated quickly since it is in cache, so CPU will fetch $\mathit{array}2[\mathit{array}1[x] \cdot 4096]$ to cache, and while