Mestrado em Engenharia Informática (MEI)
Mestrado Integrado em Engenharia Informática (MiEI)

Perfil de Especialização **CSI** : Criptografia e Segurança da Informação

Engenharia de Segurança

jose.miranda@devisefutures.com

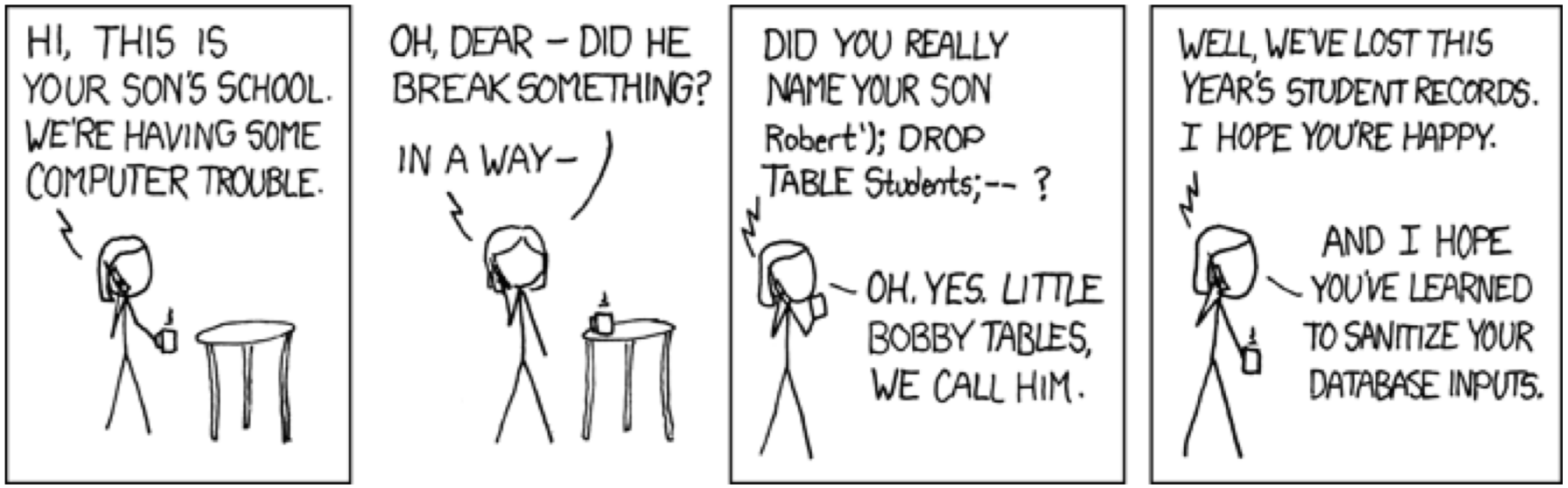
Engenharia de Segurança Project

- Report and other data (source code, settings, ...) to be submitted until 19/Jun/2019
- Presentation to be made until 26/Jun/2019
- Until June 26, 2019, each member of the Working Group will send the 360 analysis of the elements of his Group by mail, evaluating the work of each member of the Working Group (including his own) from 0 - 100.

Please note that the presentation date must be at least one week after the report is submitted.

Topics

- *Input* Validation



Input Validation

- **All input** to a program (keyboard, network, files, external devices, environment variables, web services, ...) can be the **source of security vulnerabilities** and bugs;
- Any program that processes **input data without proper validation** is susceptible to **security vulnerabilities**;
- An **attacker** can pass **malformed arguments** to any program input parameter;
- All **input** must be **treated** as **potentially hazardous**.
- These issues are particularly relevant in programs with setuid root permissions (i.e., users run the program as if they were root), or programs that run in privileged mode.

Input Validation

CVE ID	Vulnerability type	Publish Date	CVSS Score	Description
CVE-2019-9918	SQL Injection	03/29/2019	9.1	An issue was discovered in the Harmis JE Messenger component 1.2.2 for Joomla!. Input does not get validated and queries are not written in a way to prevent SQL injection. Therefore arbitrary SQL-Statements can be executed in the database.
CVE-2019-9712	Cross-Site Scripting (XSS)	03/12/2019	6.1	An issue was discovered in Joomla! before 3.9.4. The JSON handler in com_config lacks input validation, leading to XSS.
CVE-2019-9117	OS Command Injections	03/07/2019	9.8	An issue was discovered on Motorola C1 and M2 devices with firmware 1.01 and 1.07 respectively. This issue is a Command Injection allowing a remote attacker to execute arbitrary code, and get a root shell. A command Injection vulnerability allows attackers to execute arbitrary OS commands via a crafted /HNAP1 POST request
CVE-2019-6781	Injection	05/17/2019	7.5	An Improper Input Validation issue was discovered in GitLab Community and Enterprise Edition before 11.5.8, 11.6.x before 11.6.6, and 11.7.x before 11.7.1. It was possible to use the profile name to inject a potentially malicious link into notification emails.
CVE-2019-6210	Buffer Errors	03/05/2019	7.8	A memory corruption issue was addressed with improved input validation. This issue is fixed in iOS 12.1.3, macOS Mojave 10.14.3, tvOS 12.1.2, watchOS 5.1.3. A malicious application may be able to execute arbitrary code with kernel privileges.
CVE-2019-6318	Input Validation	04/11/2019	9.8	HP LaserJet Enterprise printers, HP PageWide Enterprise printers, HP LaserJet Managed printers, HP Officejet Enterprise printers have an insufficient solution bundle signature validation that potentially allows execution of arbitrary code.

Input Validation

- **Attack surface** of an application consists of the set of interfaces through which inputs can be received from outside:
  - Remote communication mechanisms (e.g., sockets, web services, ...);
  - Mechanisms of communication between processes (e.g., signals, ...);
  - Programmatic application interface (API);
  - Files used by the application;
  - Interface used (e.g., application arguments, user input, ...);
  - Operating system (e.g., environment variables, ...).

- Golden rule in building secure software is to **never trust on input**.

Input validation from the parent process

- In **Unix** family operating systems, **applications** are typically launched from a *shell* and **run as a child process** of that ***shell***.
- **Attacker** with access to that *shell*, can launch the application, **providing input that exploits some vulnerability**:
  - Arguments with undue size (may cause buffer overflow);
  - Signal processing routines with malicious code;
  - Define the default permissions of the files created by the application (through the umask command);
  - Environment variables with erroneous values.
    - PATH environment variable stores directories where executable programs may be stored (when a program is asked to run without specifying its absolute path, the operating system traverses the directories stored in the PATH in the order they appear in that variable, until it finds the program and executes it).

Input validation from the parent process

Example:

- C functions ***system****(command)* and ***popen****(command, type)* execute the program/command passed as argument, with the environment variables of the parent process → **avoid both functions**

- Suppose that a program includes the instruction **system("ls")** to list the contents of the current directory (note that the legitimate *ls* program is stored in the /bin directory)
- What would happen if an attacker:
  - Changed the value of the PATH to".:/usr/local/bin:/usr/bin:/bin", and
  - Performed the bash command "cp ./programa_malicioso ./ls" ?
- And what would happen if the program had *setuid root* permissions?

Input validation from the parent process

- What other environment variables are used by your program or by the libraries/APIs you use?
- If the libraries/APIs you use do not properly control the environment variables:
  - It is up to you to carry out this control, or
  - Define the variable yourself.

Input Validation: Metacharacters

- **Metacharacters** are a source of particular concern since they are **responsible for a large number of vulnerabilities**, typically in applications that handle *strings*.
- **Solution** to this type of vulnerability is simple: validate the incoming inputs, controlling the accepted (meta)characters
  - Opt for **white listing** techniques where a list with the valid characters accepted by the application is defined.
  - Do not opt for **black listing**, i.e. by the list of characters that should not be accepted by the application. Why?
    - For example, if UTF-8 encoding is being used, the '.' (dot) character can also be written as '%2e' and '%c0%ae', among others.

Input Validation: Metacharacters

- Three types of most common Metacharacter-based attacks:
  - Embedded delimiters
    - When the input to the application includes different types of information separated by delimiters.
    - Suppose an application that stores user names with their passwords in a file where each line has the format *user:password\n*
    - What can happen if Alice changes the password, to *potato\nhacker:ola123* ?

Input Validation: Metacharacters

- Três tipos de ataques mais comuns baseados em Metacaracteres:
  - Embedded delimiters
  - Injection of character \0
    - Dangerous because it is not always interpreted as a string terminator (for example, in Perl and Java, as opposed to C/C ++)
    - Consider a Web application built in C that allows you to open text files with a .txt extension.
    - What happens if Alice supplies the /etc/passwd\0.txt string as the file name?

Input Validation: Metacharacters

- Três tipos de ataques mais comuns baseados em Metacaracteres:
  - Embedded delimiters
  - Injection of character \0
  - Injection of separators
    - Injection of command separators, which can allow execution of arbitrary commands. In Unix, using the metacharacter ';'.
    - Injection of folder separators, usually called as *path traversal attack,* can allow reading and/or writing arbitrary files.
      - Consider a web application that given a user prints statistics using *system("cat", "/var/stats/$username");*
      - How can Alice take advantage of this vulnerability and print any file on the system, for example the /etc/passwd file?

Input Validation: Format String Vulnerability

- Class of vulnerabilities in which:
  - A validação de entradas necessária para evitar a vulnerabilidade é extremamente simples.
  - The **lack of validation of inputs allows an attacker to control the execution of an application**;
  - The input validation required to avoid vulnerability is extremely simple.
- Most prevalent and dangerous class of vulnerabilities in C and C++, although other languages (e.g., Java, Perl, PHP, Python, and Ruby) also allow formatting strings with related vulnerabilities.

Input Validation: Format String Vulnerability

- Simple (and classic) example of format string vulnerability:

```c
#include <stdio.h>

int main(int argc, char **argv) {
        char buf[1024];

        if(argc > 1) {
                strncpy(buf, argv[1], 1023);
                buf[1023] = '\0';
                printf(buf);
        }
}
```

- The first argument of the printf function is under control of the user of the application, and can be used to specify the format of different data types (e.g., %d indicates an integer variable, %s a string, ...)

```
$ ./a.out  "string - %s | apontador -  %p | inteiro -  %d"
string - (null) | apontador -  0xffffffffffffffff | inteiro -  19
```

Input Validation: Format String Vulnerability

- Simple (and classic) example of format string vulnerability:

```c
#include <stdio.h>

int main(int argc, char **argv) {
        char buf[1024];

        if(argc > 1) {
                strncpy(buf, argv[1], 1023);
                buf[1023] = '\0';
                printf(buf);
        }
}
```

- Whenever the format string can be controlled by an attacker, this is a format string vulnerability.
  - It occurs in all families of functions that have format strings as argument (eg, printf, err, syslog).

Input Validation: Format String Vulnerability

- Simple (and classic) example of format string vulnerability:

```
1   #include <stdio.h>
2
3   int main(int argc, char **argv) {
4           char buf[1024];
5
6           if(argc > 1) {
7                   strncpy(buf, argv[1], 1023);
8                   buf[1023] = '\0';
9                   printf(buf);
10          }
11  }
```

- How to resolve this vulnerability?
  - Replacing in line 9 *printf(buf)* by *printf("%s", buf)* .

```
$ ./a.out  "string - %s | apontador -  %p | inteiro -  %d"
string - %s | apontador -  %p | inteiro -  %d
```

Input Validation: Format String Vulnerability

- Simple (and classic) example of format string vulnerability:

```c
#include <stdio.h>

int main(int argc, char **argv) {
        char buf[1024];

        if(argc > 1) {
                strncpy(buf, argv[1], 1023);
                buf[1023] = '\0';
                printf(buf);
        }
}
```

- What is the biggest problem of this vulnerability?
  - Allows read / write stack values through the appropriate use of string formatters.
  - Ex: (the value 41 hexadecimal corresponds to the value 65 in decimal, which corresponds to the letter 'A').

```
$ ./a.out AAAAAAAA%p%p%p%p%p%p%p%p%p%p
AAAAAAAA0x1d0x7ffeea8db9400x1d0x00xffffffff000000000x00x7ffeea8db3700x7ffeea8db7a80x20x4141414141414141
```

Input Validation

- Risk
  - If **input data** are not **validated** to ensure they contain the **type**, **size**, and **correct structure of information**, problems can (and will) happen;
  - **Input validation errors** can lead to *buffer overflows* if the data is used as indexes for an array, or used as a base of SQL injection allowing to access/change/delete private data in a Database;
  - **Attackers** can **use carefully chosen inputs** in order to **cause arbitrary code execution**. This technique can be used to erase data, cause damage, propagate worms, or obtain confidential information

Input Validation

- "Responsible" validation of **all** input
  - Type: validate that the **input has the expected data type**, for example age is *int*. Many programs deal with input data by assuming that it is a string, then verifying that this string contains the appropriate characters, and converting it to the desired data type;
  - Size: Validate that the **input has the expected size** (for example, the phone number has 9 digits);
  - Interval: validate that the **input is within the expected range** (for example, the value of the month is between 1 and 12);
  - Reasonability: Validate that the **input has a reasonable value** (for example, the name does not contain punctuation characters or non-alphanumeric characters);
  - Division by Zero: validate the input so as not to accept values that may cause problems later in the program, such as division by zero;
  - Format: Validate that the **input data is in the proper format** (for example, the date is in DD/MM/YYYY format);
  - Mandatory data: ensure that the user enters the mandatory data;
  - *Checksums*: many ID numbers have check digits (additional digits entered at the end of the number for validation). See check digit of Cartão de Cidadão, Passport, credit cards, Luhn algorithm.

Input Validation

- Use the programming language tools
  - Languages such as C and C++ read (by default) the input to a character buffer, without validating the buffer limit, causing buffer overflow and input validation problems. However, there are more robust reading libraries specifically designed for security;
  - Strongly typed languages, such as Java and C++, require that the type of data stored in a variable be known a priori (which leads to type incompatibilities when, for example, a string is inserted in response to an integer request);
  - Untyped languages such as PHP or Python do not have these requirements - any variable can hold any value. This does not eliminate the validation problems (test the input of a string to be used as index of an array);
- Proper Recovery
  - A robust program must treat an invalid input in an appropriate, correct, and safe manner, by repeating the input request or by continuing with predefined values (truncating or reformatting data to fit the intended input should be avoided).