Politecnico di Milano
Dip. Elettronica e Informazione
Milano, Italy

Introduction to secure programming

Secure programming

- Good software engineering states that a program must match its specifications, which in turn must satisfy requirements
  - Functional requirements: the software must accomplish what it was designed to do
  - Nonfunctional requirements: “ease of use”, “safety”, “security” …
- Creating inherently secure applications is a fundamental, yet often unknown, skill for a good developer or software engineer
  - Proof: the sheer number of vulnerabilities in today's software

What are vulnerabilities?

- Programs should behave according to specifications; if they don't, it is a *bug*
- If the violated specification is a security specification, then it is a *vulnerability*
- A way to use a vulnerability to violate a security policy (or more in general to violate the CIA paradigm applied to a given system) is called an *exploit*
- Let's try to imagine a simple example of a vulnerability and its exploit

Let us take an example

- Let's use a UNIX-like system such as Linux for simplicity
- In such systems, programs can have a saved user ID (SUID). In this case the program is executed with the UID of the user, until it raises its privileges to the saved ones
- Let us take as an example a program which is SUID root and uses a text config file
- Let us suppose that there is a config switch on the command line:

```
./program --config /etc/my-config
```

- Let us think of a privilege escalation scenario: an unprivileged user wants to become root

How to get it all wrong

ALGORITHM 1

- Start
- **EUID -> SUID**
- read(config)
- r = parse(config)
- IF r = OK do_things()
  ELSE error("...")
- exit()

```
[user@host]$ ./program
  --config /etc/shadow

ERROR in config file,
  line 1:

root:<password>:….
```

ALGORITHM 2

- Start
- read(config)
- **EUID -> SUID**
- r = parse(config)
- IF r = OK do_things()
  ELSE error("...")
- exit()

```
[user@host]$ ./programma
  --config /etc/shadow

ERROR: config file not
  found, or unable to
  read
```

Key issues in "secure design"

- Reducing privileged parts to a minimum
- KISS (Keep It Simple, Stupid)
- Discard privileges as soon as possible
- Open design: just as with Kerchoffs principle, the programme must not rely on obscurity for security
- Concurrency and race conditions
- Remote procedure calls (incl. RPC, CORBA, RMI, Web Services, etc.)

Key issues in “secure programming”

- Fail-safe and default deny
- Usage of shared resources (e.g. mktemp)
- Usage of libraries of unknown security
- **Filter the INPUT**
- Filter the output as well!
- Do not write any crypto, password and secret management code, use trusted code which has been audited already
- Use trusted entropy sources such as /dev/urandom

Code security by examples

- We will see four main examples of secure programming and insecure programming
  - Standard applications
    - Buffer Overflows
    - String format bugs
  - Web applications
    - SQL Injection bugs
    - Cross site scripting bugs
- There are many other examples! We will just deal with a few cases