Software security, secure programming

Lecture 1: introduction

Master M2 Cybersecurity & MoSiG

Academic Year 2020 - 2021

Who are we ?

Teaching staff

- Laurent Mounier (UGA), Marie-Laure Potet (G-INP)
  Mathias Ramparon (G-INP)
- research within Verimag Lab
- research focus: formal verification, code analysis, compilation techniques, language semantics ... and **(software) security** !

Attendees

- Master M2 on Cybersecurity [mandatory course]
- Master M2 MoSiG [optionnal course]

$\rightarrow$ various skills, backgroud and interests . . .

Agenda

Part 1: an overview of software security and secure programming

- 6 weeks (18 hours)
- for all of you . . .
- classes on **tuesday** (11.15am-12.45am) and **wednesday** (2pm-3.30pm)
  $\rightarrow$ includes lectures, exercises and *lab sessions* . . .

Part 2: some tools and techniques for software security

- 7 weeks (21 hours)
- for M2 Cyberscurity only
- class on **tuesday** (8.15am - 11.15am)

Examination rules

The rules of the game ...

Assignments

- M_1: a written exam (duration $\sim$ 1.5h, mid-November)
- M_2: a (short) report on lab sessions
- M_3: an oral presentation (in January)
- M_4: final written exam (duration=2h, end of January)

Mark computation (3 ECTS)

- for MoSiG students:

$$M = (0.3 \times M_2) + (0.7 \times M_1)$$

- for Cybersecurity students:

$$M = (0.5 \times M_1 + 0.25 \times M_2 + 0.25 \times M_3)/2 + (0.5 \times M_4)$$

Course user manual

An (on-going) course web page . . .

`http://www-verimag.imag.fr/~mounier/Enseignement/Software_Security`

- course schedule and materials (slides, etc.)
- weekly, reading suggestions, to complete the lecture
- other background reading/browsing advices . . .

During the classes . . .

Alternation between lectures, written excercices, lab exercises . . .
. . . but no “formal” lectures → questions & discussions always welcome !

heterogeneous audience + open topics ⇒ **high interactivity level !**

Prerequisites
Ideally . . .

This course is concerned with:

Programming languages

- at least one (classical) imperative language:
  C or C++ ? Java ?? Python ??? . . .
- some notions on compilation & language semantics

What happens behind the curtain

Some notions about:

- assembly code (ARM, x86, others . . .)
- memory organization (stack, heap)

Outline

Some practical information

What **software security** is (not) about ?

About software security

The context: computer system security ...

Question 1: what is a "computer system", or an **execution plateform** ?

Many possible incarnations, e.g.:

- (classical) computer: mainframe, server, desktop
- mobile device: phone, tablets, audio/video player, etc. ...up to IoT, smart cards, ...
- embedded (networked) systems: inside a car, a plane, a washing-machine, etc.
- cloud computing, virtual execution environment
- but also industrial networks (Scada), ...etc.
- and certainly many more !

→ 2 main characteristics:

- include hardware + **software**
- open/connected to the **outside world** ...

The context: computer system security . . . (ct'd)

Question 2: what does mean **security** ?

- a set of general **security** properties: CIA
  Confidentiality, Integrity, Availability (+ Non Repudiation + Anonymity + . . .)

- concerns the **running** software + the whole **execution plateform**
  (other users, shared resources and data, peripherals, network, etc.)

- depends on an intruder model
  $\rightarrow$ there is an "external actor"[1] with an **attack objective** in mind, and able to elaborate a dedicated strategy to achieve it ($\neq$ hazards)
  $\hookrightarrow$ something beyond **safety** and **fault-tolerance**

$\rightarrow$ A possible definition:

- functionnal properties = what the system should do
- security properties = what it should **not allow** w.r.t the intruder model . . .

Rk: functionnal properties do matter for "security-oriented" software (firewalls, etc.)!

[1]could be the user, or the **execution plateform itself!**

Example 1: password authentication

Is this code “secure” ?

```
boolean verify (char[] input, char[] passwd , byte len) {
  // No more than triesLeft attempts
  if (triesLeft < 0) return false ; // no authentication
  // Main comparison
  for (short i=0; i <= len; i++)
    if (input[i] != passwd[i]) {
       triesLeft-- ;
       return false ;  // no authentication
    }
  // Comparison is successful
  triesLeft = maxTries ;
  return true ;  // authentication is successful
}
```

functional property:

$$\texttt{verify}(\texttt{input}, \texttt{passwd}, \texttt{len}) \Leftrightarrow \texttt{input}[0..\texttt{len}] = \texttt{passwd}[0..\texttt{len}]$$

What do we want to protect ? Against what ?

- confidentiality of `passwd`, *information leakage ?*
- no unexpected runtime behaviour
- code integrity, etc.

Example 2: file compression

Let us consider 2 programs:

- `Compress`, to compress a file f
- `Uncompress`, to uncompress a (compressed) file c

A functional property: **the one we will try to validate . . .**

$$\forall f.\texttt{Uncompress}(\texttt{Compress}(f)) = f \quad \textbf{(1)}$$

But, what about uncompressing an arbitrary (i.e., *maliciously crafted*) file ?
(e.g., CVE-2010-0001 for `gzip`)

A security property:

$$(\nexists f.\texttt{Compress}(f) = c) \Rightarrow (\texttt{Uncompress}(c) \not\rightsquigarrow$$  $)$

(uncompressing an arbitrary file should not produce unexpected **crashes***)*

Actually **(2)** is much more difficult to validate than **(1)** . . .

Demo: `make ‘python -c ’print "A"*5000’ ‘`

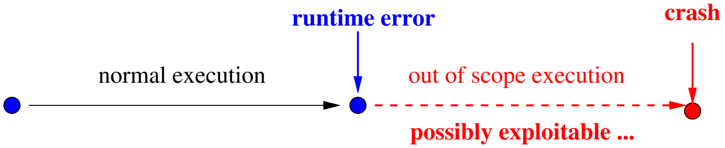
Why do we need to bother about crashes ?

crash = consequence of an unexpected run-time error

- not trapped/foreseen by the programmer
- nor at the "programming language level"

⇒ part of the execution:

- may take place outside the program scope (beyond the program semantic)
- but can be controled/exploited by an attacker (∼ ***"weird machine"***)



↪ may **break** all security properties ...
from simple denial-of-service to **arbitrary code execution**

Rk: may also happen **silently** (without any crash !)

Some (not standardized) definitions . . .

Bug: an error (or defect/flaw/failure) introduced in a SW, either

- at the specification / design / algorithmic level
- at the programming / coding level
- or even by the compiler (or any other pgm transformation tools) . . .

Vulnerability: a weakness (for instance a bug !) that opens a security breach

- **non exploitable** vulnerabilities: there is no (known !) way for an attaker to use this bug to corrupt the system
- **exploitable** vulnerabilities: this bug can be used to elaborate an attack (i.e., write an exploit)

Exploit: a concrete program input allowing to *exploit* a vulnerability
(from an attacker point of view !)
PoC exploit: assumes that existing protections are disabled
(i.e., they can be hijacked wit other existing exploits)

Malware: a piece of code "injected" inside a computer to corrupt it
$\rightarrow$ they usually exploit existing vulnerabilities . . .

Software vulnerability examples

Case 1 (not so common . . .)

Functional property not provided by a security-oriented component

- too weak crypto-system,
- no (strong enough) authentication mechanism,
- etc.

Case 2 (**the vast majority !**)

Insecure coding practice in (any !) software component/application

- improper input validation $\rightsquigarrow$ SQL injection, XSS, etc.
- insecure shared resource management (file system, network)
- information leakage (lack of data encapsulation, side channels)
- exploitable run-time error
- etc.

The intruder model

Who is the attacker ?

- a malicious external user, interacting via regular input sources
  e.g., keyboard, network (man-in-the-middle), etc.
- a malicious external "observer", interacting via side channels
  (execution time, power consumption)
- another application running on the same plateform
  interacting through shared resources like caches, processor elements, etc.
- the execution plateform itself (e,g., when compromised !)

What is he/she able to do ?

At low level:

- unexpected memory read (data or code)
- unexpected memory write (data or code)

$\Rightarrow$ powerful enough for

- information disclosure
- unexpected/arbitrary code execution
- priviledge elevation, etc.

Outline

Some practical information

What **software security** is (not) about ?

About software security

Some evidences regarding cyber (un)-security

So many examples of successful computer system attacks:

- the **"famous ones"**: (at least one per year !)
  Morris worm, Stuxnet, Heartbleed, WannaCry, Spectre, etc.
- the **"cyber-attacks"** against large organizations: (+ 400% in 10 years)
  Sony, Yahoo, Paypal, e-Bay, etc.
- all the daily **vulnerability alerts**: [have a look at these sites !]
  `http://www.us-cert.gov/ncas`
  `http://www.securityfocus.com`
  `http://www.securitytracker.com`
- etc.

Why ? Who can we blame for that ??

- $\nexists$ well defined recipe to build secure cyber systems in the large
- permanent trade-off beetween **efficiency** and **safety/security**:
  - HW and micro-architectures (**sharing** is everywhere !)
  - operating systems
  - programming languages and applications
  - coding and software engineering techniques

But, what about **software** security ?

Software is **greatly involved** in "computer system security":

- it plays a major role in **enforcing security properties**:
  crypto, authentication protocols, intrusion detection, firewall, etc.
- but it is also a major source of **security problems**[2] ...
  "90 percent of security incidents result from exploits against defects in software" (U.S. DHS)

→ SW is clearly one of the weakest links in the security chain!

Why ???

- we do not no very well how to write **secure** SW
  we do not even know how to write **correct** SW!
- behavioral properties can't be validated on a (large) SW
  impossible by hand, untractable with a machine
- programming languages not designed for security enforcement
  most of them contain numerous traps and pitfalls
- programmers feel not (so much) concerned with security
  security not get enough attention in programming/SE courses
- heterogenous and nomad applications favor unsecure SW
  remote execution, mobile code, plugins, reflection, etc.

[2] outside security related code!

Some evidences regarding software (un)-security (ct'd)

An increasing activity in the "defender side" as well ...

- all the daily security patches (for OS, basic applications, etc.)
- companies and experts specialized in software security
  code audit, search for 0days, malware detection & analysis, etc.
  "bug bounties" [`https://zerodium.com/program.html`]
- some important research efforts
  from the main software editors (e.g., MicroSoft, Google, etc)
  from the academic community (numerous dedicated conferences)
  from independent "ethical hackers" (blogs, etc.)
- software verification tools editors start addressing security issues
  e.g.: dedicated static analyser features
- international cooperation for vulnerability disclosure and classification
  e.g.: CERT, CVE/CWE catalogue, vulnerability databases
- government agencies to promote & control SW security
  e.g.: ANSSI, Darpa "Grand Challenge", etc.

Couter-measures and protections (examples)

Several existing mechanisms to **enforce** SW security

- at the programming level:
  - disclosed vulnerabilities → language weaknesses databases
    ↪ *secure* coding patterns and libraries
  - aggressive compiler options + code instrumentation
    ↪ early detection of unsecure code

- at the OS level:
  - sandboxing
  - address space randomization
  - non executable memory zones
  - etc.

- at the hardware level:
  - Trusted Platform Modules (TPM)
  - secure crypto-processor
  - CPU tracking mechanims (e.g., Intel Processor Trace)
  - etc.

Techniques and tools for assessing SW security

Several existing mechanisms to **evaluate** SW security

- code review . . .
- fuzzing:
  - run the code with “unexpected” inputs $\rightarrow$ pgm crashes
  - (tedious) manual check to find exploitable vulns . . .
- (smart) testing:
  coverage-oriented pgm exploration techniques
  (genetic algorithms, dynamic-symbolic executions, etc.)
  + code instrumentation to detect (low-level) vulnerabilities
- static analysis: approximate the code behavior to detect **potential** vulns
  ($\sim$ code optimization techniques)

In practice:

- only the binary code is **always available** and **useful** . . .
- **combinations** of all these techniques . . .
- exploitability analysis still challenging . . .

Course objectives (for the part 1)

Understand the root causes of common weaknesses in SW security

- at the programming language level
- at the execution platform level

$\rightarrow$ helps to better choose (or deal with) a programming language

Learn some methods and techniques to build more secure SW:

- programming techniques:
  languages, coding patterns, etc.
- validation techniques:
  what can(not) bring existing tools ?
- counter-measures and protection mechanisms

Course agenda (part 1)

See

`http://www-verimag.imag.fr/~mounier/Enseignement/Software_Security`

Credits:

- E. Poll (Radboud University)
- M. Payer (Purdue University)
- E. Jaeger, O. Levillain and P. Chifflier (ANSSI)