UE SCLAM

Sécurité Logicielle

Lecture 1: introduction

Master M2 Cybersécurité et Informatique Légale

Academic Year 2023 - 2024

Who are we ?

Teaching staff

- Laurent Mounier (UGA)
- research within Verimag Lab
- research focus: formal verification, code analysis, compilation techniques, language semantics ... and **(software) security** !

Attendees

- Master M2 CSI students

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

UE SCLAM ?

Sécurité des Composants et des Logiciels et Applications Multimédia

- 42 heures de cours/TD et 37 heures de TP/TD
- $\sim$ 4 intervenants
- Thèmes couverts :
  - Sécurité Logicielle
  - Sécurité Matérielle
  - Rétro-ingéniérie
  - Sécurité des Systèmes Embarqués
  - etc.
- Thème “Sécurité Logicielle”
  - an overview of software security and secure programming
  - some tools and techniques for software security

Evaluation de l'UE

Les règles du jeu . . .

Plusieurs notes possibles

- compte-rendus de TP
- (courtes) présentations orales
- mini "projets"
- Quizz, QCMs, etc.

$\rightarrow$ une **moyenne pondérée** de l'ensemble de ces notes . . .

Course user manual

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

`https://im2ag-moodle.univ-grenoble-alpes.fr/course/view.php?id=367`

- 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 exercices, lab exercises . . .

. . . but no “formal” lectures $\rightarrow$ questions & discussions always welcome !

heterogeneous audience + open topics $\Rightarrow$ **high interactivity level !**

Prerequisites

Ideally . . .

This course is concerned with:

Programming languages

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

What happens behind the curtain

Some notions about:

- assembly code (ARM, x86, others . . .)
- runtime memory layout (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, laptop, etc.
- 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/remote 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:

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

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: $\forall c.\texttt{Uncompress}(c) \not\rightarrow$ 

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

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

(out-dated) Demo: `make 'python -c 'print "A"*5000''`

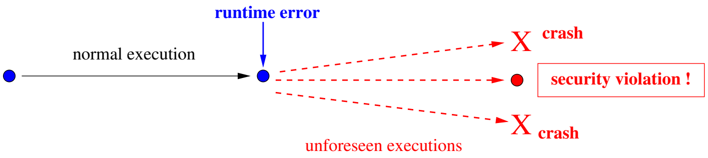
Why do we need to bother about crashes ?

crash = consequence of an unexpected run-time error

- not foreseen by the programmer and compiler ...
- ...and not (always) accurately trapped at runtime

⇒ some part of the execution:

- may take place outside the program scope
  (not following the regular 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)
- **0-day** vulnerabilities: yet unpublished (hence **not patched !**)

Exploit: a concrete attacker behavior allowing to:

1. trigger a (sequence of) vulnerability(-ies)
2. leading to a security property violation

Ex: a single program input, or a complex sequence of interactions with the target program and/or its execution environment . . .

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 $\leadsto$ 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 never-ending records of **“cyber-attacks”** against large organizations (private companies, public (infra-)structures)
- all the daily **vulnerability alerts**: [have a look at these sites !]
  `https://cve.mitre.org/`
  `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 cyber (un)-security

A recent CVE example (sept 26th 2023):

Vulnerability Details : CVE-2023-4259

Two potential buffer overflow vulnerabilities at the following locations in the Zephyr eS-WiFi driver source code.

Vulnerability category: Overflow

Published 2023-09-26 00:15:11 Updated 2023-09-26 00:15:11 Source Zephyr Project

View at NVD, CVE.org

Exploit prediction scoring system (EPSS) score for CVE-2023-4259

We don't have an EPSS score for this CVE yet EPSS FAQ

CVSS scores for CVE-2023-4259

Base Score	Base Severity	CVSS Vector	Exploitability Score	Impact Score	Source
7.1	HIGH	CVSS:3.1/AV:A/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:H	2.8	4.2	vulnerabilities@zephyrproject.org

CWE ids for CVE-2023-4259

CWE-120 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
The product copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.
Assigned by: vulnerabilities@zephyrproject.org (Secondary)

CWE-193 Off-by-one Error
A product calculates or uses an incorrect maximum or minimum value that is 1 more, or 1 less, than the correct value.
Assigned by: vulnerabilities@zephyrproject.org (Secondary)

References for CVE-2023-4259

https://github.com/zephyrproject-rtos/zephyr/security/advisories/GHSA-gghm-c696-f4j4

Numbers of CVEs per year

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

`https://im2ag-moodle.univ-grenoble-alpes.fr/course/view.php?id=545`

Credits:

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