Software security, secure programming

Lecture 1: introduction

Master M2 Cybersecurity & MoSiG

Academic Year 2019 - 2020
Who are we?

Teaching staff

- Laurent Mounier (UGA) & Marie-Laure Potet (Grenoble-INP)
- research within Verimag Lab (PACSS team)
- research focus: formal verification, code analysis, compilation techniques, language semantics ... and (software) security!

Attendees

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

→ various skills, background and interests ...
Agenda

Part 1: an overview of software security and secure programming

- 6 weeks (18 hours)
- for all of you...
- classes on **tuesday** (11.30am-1pm) and **wednesday** (1.30.pm-3pm)
  → 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:
  \[ M = (0.3 \times (M_1 + M_3)/2) + (0.2 \times M_2) + (0.5 \times M_4) \]
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 excercises, 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 an execution platform?
The context: computer system security . . .

Question 1: what is 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!
Question 1: what is an execution platform?

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 interesting characteristics:
  - include hardware + software
  - open/connected to the outside world ...
The context: computer system security . . . (ct’d)

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

---

1 could be the user, or the *execution platform itself!*
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 platform
    (other users, shared resources and data, peripherals, network, etc.)

▶ depends on an intruder model
    → there is an “external actor”\(^1\) with an attack objective in mind, and
    able to elaborate a dedicated strategy to achieve it (≠ hazards)
    ← something beyond safety and fault-tolerance

→ 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 platform itself!
Example 1: file compression

Let us consider 2 programs:
- \texttt{Compress}, to compress a file \( f \)
- \texttt{Uncompress}, to uncompress a (compressed) file \( c \)

Expected behavior: \textbf{the one we will try to validate}!

\[ \forall f. \text{Uncompress}(\text{Compress}(f)) = f \quad (1) \]
Example 1: file compression

Let us consider 2 programs:

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

Expected behavior: the one we will try to validate!

\[
\forall f. \text{Uncompress(Compress}(f)) = f \quad (1)
\]

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

\[
(\forall f. \text{Compress}(f) = c) \Rightarrow (\text{Uncompress}(c) \not\Rightarrow)
\]

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

**Demo:** `make 'python -c 'print "A"*5000'`
Example 2: Pin Verifier

Is this code “secure”?

boolean verify (byte[] buffer, short ofs, byte len)
{
    // No comparison if PIN is blocked
    if (triesLeft < 0) return false;
    // Main comparison
    for (short i=0; i < len; i++)
        if (buffer[ofs+i] != pin[i]) {
            triesLeft--;
            authenticated = false;
            return false;
        }
    // Comparison is successful
    triesLeft = maxTries;
    authenticated = true;
    return true;
}
Example 2: Pin Verifier

Is this code “secure”? 

```java
boolean verify (byte[] buffer, short ofs, byte len)
{
    // No comparison if PIN is blocked
    if (triesLeft < 0) return false;
    // Main comparison
    for (short i=0; i < len; i++)
        if (buffer[ofs+i] != pin[i]) {
            triesLeft--;
            authenticated = false;
            return false;
        }
    // Comparison is successful
    triesLeft = maxTries;
    authenticated = true;
    return true;
}
```

What do we want to protect? Against what?
Example 2: Pin Verifier

Is this code “secure”? Yes

boolean verify (byte[] buffer, short ofs, byte len)
{
    // No comparison if PIN is blocked
    if (triesLeft < 0) return false;
    // Main comparison
    for (short i=0; i < len; i++)
        if (buffer[ofs+i] != pin[i]) {
            triesLeft--;
            authenticated = false;
            return false;
        }
    // Comparison is successful
    triesLeft = maxTries;
    authenticated = true;
    return true;
}

What do we want to protect? Against what?

➡ no unexpected runtime behaviour
➡ confidentiality of pin, information leakage?
➡ code integrity, etc.
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.
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 ??
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 ??

- ∅ well defined recipe to build secure cyber systems in the large
- permanent trade-off between 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**

“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 ???

\(^2\) outside security related code!
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**

  “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 ...
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.)
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 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.)
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
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
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.
Outline

Some practical information

What software security is (not) about?

About software security
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 attacker
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 with other existing exploits)

Malware: a piece of code "injected" inside a computer to corrupt it
→ they usually exploit existing vulnerabilities . . .
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
  - there is no (known!) way for an attacker to use this bug to corrupt the system
- exploitable
  - 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 with other existing exploits)

**Malware:** a piece of code "injected" inside a computer to corrupt it

→ they usually exploit existing vulnerabilities . . .
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 attacker to use this bug to corrupt the system
- **exploitable** vulnerabilities: this bug can be used to elaborate an attack (i.e., write an exploit)
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 attacker 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)
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 attacker 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 → they usually exploit existing vulnerabilities . . .
Counter-measures and protections (examples)

Several existing mechanisms to enforce SW security
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
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.
Counter-measures and protections (examples)

Several existing mechanisms to enforce SW security

▶ at the programming level:
  ▶ disclosed vulnerabilities $\rightarrow$ language weaknesses databases
  $\leftarrow$ *secure* coding patterns and libraries
  ▶ aggressive compiler options + code instrumentation
  $\leftarrow$ 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 mechanisms (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 → 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
  (∼ code optimization techniques)
Techniques and tools for assessing SW security

Several existing mechanisms to evaluate SW security

- code review . . .

- fuzzing:
  - run the code with “unexpected” inputs → 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
  (∼ 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
  → 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)