Computer Security

Application Software Security

Prof. Jean-Noël Colin
jean-noel.colin@unamur.be
Office #306

University of Namur
Computer Science Faculty

www.unamur.be

UNIVERSITÉ DE NAMUR

Agenda

- Introduction
- Requirements engineering
- Architecture and design
- Development
- Methods and tools

Introduction

- Increasing dependency of critical functions of the society towards software systems
  - increasing interest for the attacker
- Continuously increasing size and complexity of software systems
  - higher number of vulnerabilities
  - more difficult to understand
  - more difficult to manage
- More and more frequent use of outsourcing solutions or external software
  - re-use of code and libraries
  - re-use of vulnerabilities
  - ⇒ higher level of risk
- Increasing attack complexity

Vulnerabilities

OWASP TOP 10 2013

- A1 Injection
- A2 Broken Authentication and Session Management
- A3 Cross-Site Scripting (XSS)
- A4 Insecure Direct Object References
- A5 Security Misconfiguration
- A6 Sensitive Data Exposure
- A7 Missing Function Level Access Control
- ~~A8 Cross-Site Request Forgery (CSRF)~~
- A9 Using Components with Known Vulnerabilities
- ~~A10 Unvalidated Redirects and Forwards~~

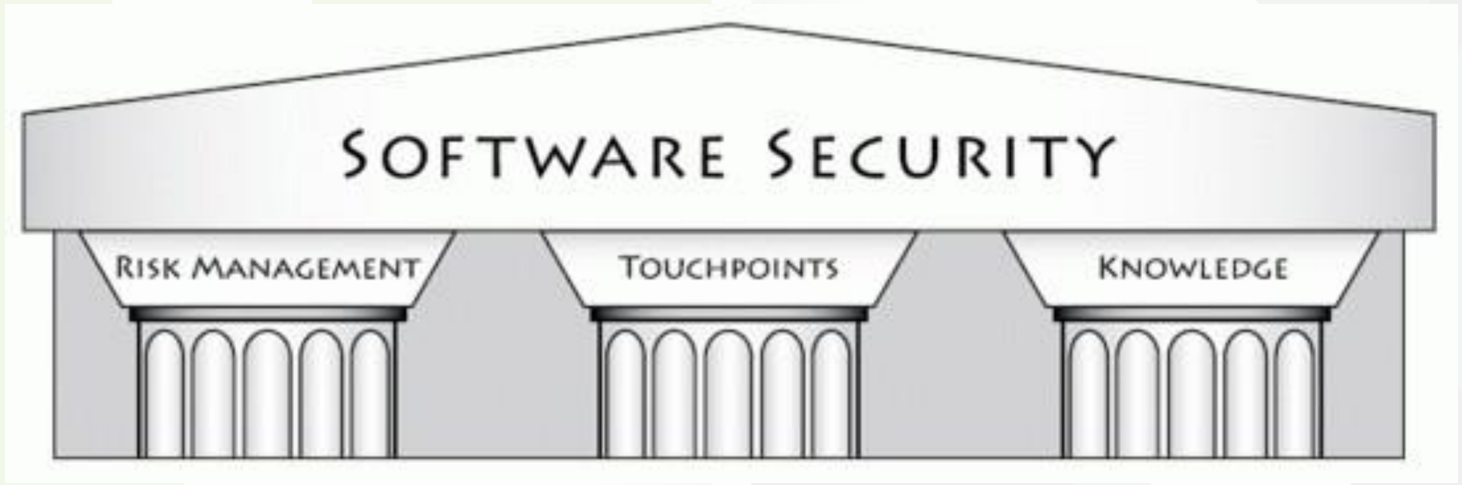
OWASP TOP 10 2017

- A1:2017-Injection
- A2:2017-Broken Authentication
- A3:2017-Sensitive Data Exposure
- A4:2017-XML External Entities (XXE) (New)
- A5:2017-Broken Access Control
- A6:2017-Security Misconfiguration
- A7:2017-Cross-Site Scripting (XSS)
- A8:2017-Insecure Deserialization (New)
- A9:2017-Using Components with Known Vulnerabilities
- A10:2017-Insufficient Logging & Monitoring (New)

Vulnerabilities

- Main vulnerability cause: **programming mistakes**
  - in particular: input/output validation
- Software security must be an integral dimension of software quality
- Security concerns must be integrated into SDLC
  - as soon as possible
  - continuously

Software Security Pillars



Source: G. McGraw, Software Security: Building Security In, Addison-Wesley, Software Security Series, 2006

What is secure software?

- Resilient to security incident
  - proper implementation of specifications
  - predictable execution
  - attack-tolerant
- Measures
  - prevention: min. #vulnerabilities, fault tolerance
  - detection: proper incident handling
  - recovery: in case of successful attack, minimize impact

Source of vulnerabilities

- Complexity
- Bad change management
- Bad use of technology
- Wrong assumptions
  - about input data
  - about environment
  - about user behavior
- Incorrect or incomplete specifications
  - missing use case, bad validation, bad error handling
- Wrong or incomplete implementation of specifications
- First target: interfaces and protocols

Managing software security

- Software security is influenced by
  - programming language
  - tools
  - operation environment
  - processes and methods used throughout SDLC
  - awareness, skills, available time of development resources
- General approach
  - continuous process
  - define and improve best practices
  - early defect detection

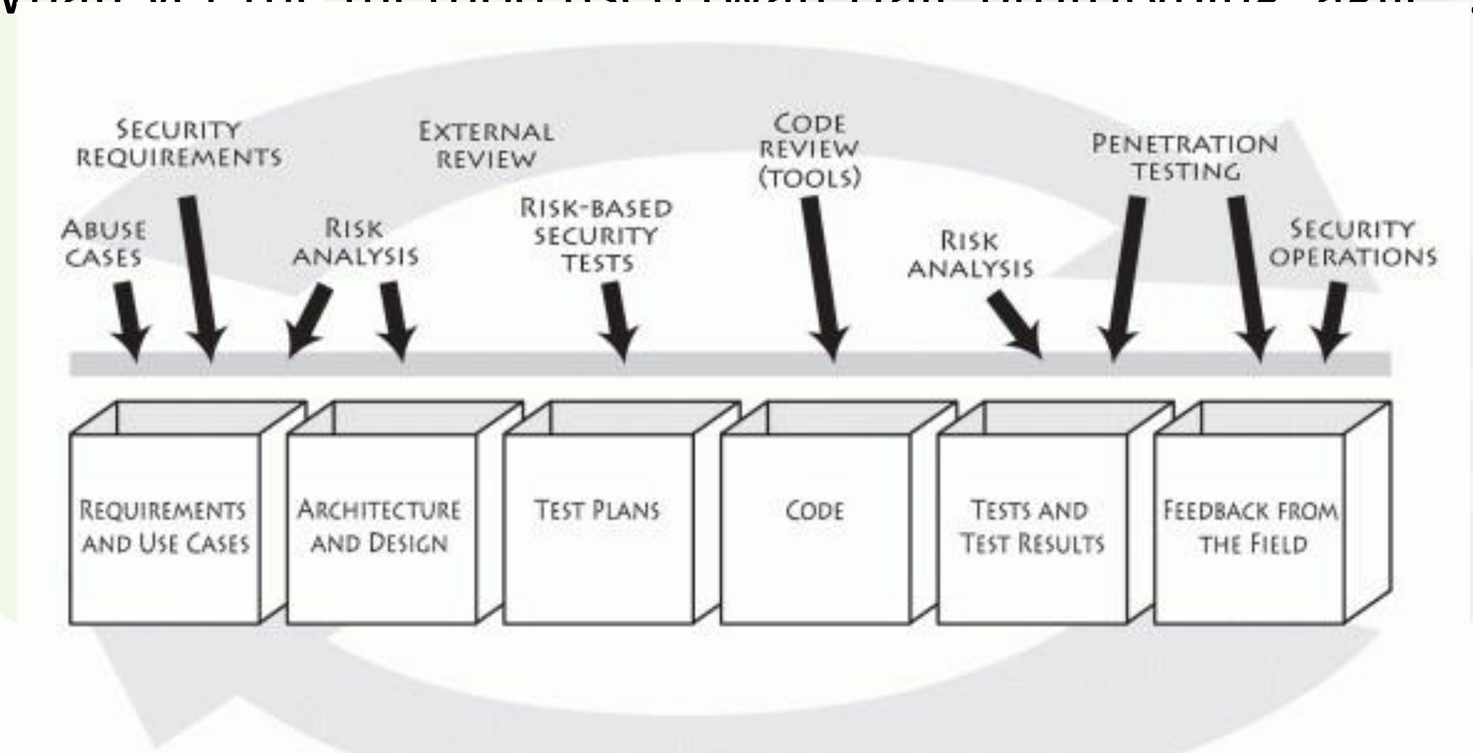
Think different!

- Secure software development requires a different mindset: adopt an adversarial perspective
  - external view: software system as a blackbox
  - when designing/developing/testing, ask yourself: what is at stake? what could be abused? what if. . . ?
  - bear in mind that software can be accessed from anywhere and that attacker can have a copy of the software
- Developer tries to close all vulnerabilities, while attacker only has to find one

“If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.” (Sun Tzu, Art of War)

Security touchpoints

- Whatever the SDLC phase (requirements, design, development, test, maintenance. . .)
- Whatever the method used (waterfall, prototyping, agile. . .)



Useful references

- Common Attack Pattern Enumeration and Classification (CAPEC)
  - http://capec.mitre.org/data/index.html
- Common Weakness Enumeration
  - http://cwe.mitre.org/
- Common Vulnerabilities and Exposure
  - http://cve.mitre.org/
- Build Security In
  - https://buildsecurityin.us-cert.gov
- Open Web Application Security Project (OWASP)
  - http://www.owasp.org
- Web Applications Security Consortium
  - http://www.webappsec.org/

Requirements engineering

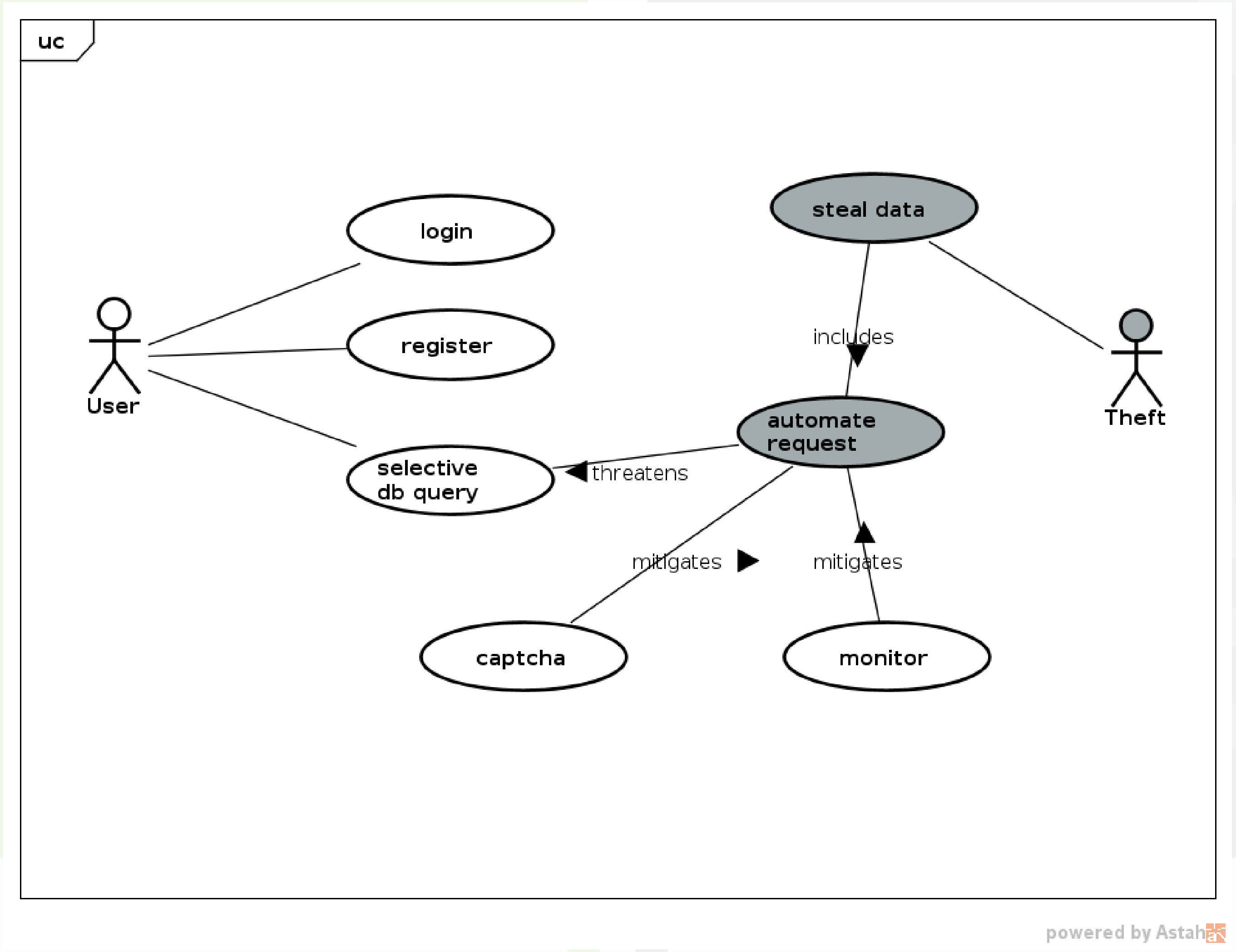
Requirements engineering

- Security is a NFR – non functional requirement; often left for later, if even considered
- Software not only has to be implemented securely, it has to offer secure functionalities, which are part of application requirements
- Different approaches, languages to capture security-related requirements
  - ex: abuse cases, KAOS Extension…

Requirements engineering

- Misuse case (Sindre, Opdahl, 2000)
  - defined as a UML extension
  - Use Case models a behavior expected from the system
  - Misuse Case models a behavior that the system must not allow
  - new concepts
    - misuser
    - misuse case
  - new relationships between Use Cases
    - ‘threatens’
    - ‘mitigates’

Misuse Cases



Architecture and design

Software Security Principles

- 50% of defects come from design errors
- main guidelines
  - as simple as possible
    - complexity = source of vulnerabilities
  - assumption: insecure and untrusted environment and user
    - do not trust, check!
  - defense in depth
    - layered approach
  - start with the weakest link

Software Security Principles

- Access control
  - least privilege
    - minimal level of privilege, for minimal duration
  - separation of duty
    - require multiple conditions to grant access
  - modular and isolated approach
    - separate functionalities in isolated modules to avoid global compromise
  - ensure all accesses are mediated by the guard
  - beware of TOCTTOU – Time Of Check To Time Of Use

Software Security Principles

- Secure error handling
  - access denied by default
  - rollback changes in case of error
  - check return values
  - define a default case (switch)
  - secure variable init value
  - minimal info disclosure (who needs to view a Java stacktrace?)
- No ‘security by obscurity’
- Mind users’ psychological acceptability

Data validation

- Principles
  - input data are always suspect
  - data cross software boundaries
    - do not pass unvalidated input to another component
- What to validate?
  - origin
  - target
  - syntax & semantic
    - String color
    - Enum color = RED, YELLOW, GREEN, BLUE;
    - Color color

Data validation

- When to validate?
  - before use, at component boundary, application entry points, message parsing
- How to validate?
  - ‘Reject known bad’
    - black list of values, patterns, formats. . .
    - is the list exhaustive?
  - ‘Accept known good’
    - white list of values, patterns, formats. . .
    - is the list exhaustive?
  - ‘Sanitization’
    - remove potentially harmful characters
    - ex: html code, LDAP query, OS command. . .
  - Tools exist!

Architecture and design

- Access concurrency
  - deadlock, loss of information, loss of integrity, unbalanced execution
  - hard to spot
  - ex. race condition
  - counter-measures
    - synchronization mechanisms (locks, semaphores. . .)
    - minimize time between check and access (TOCTTOU)
    - best practice: check, access, check

Architecture and design

- Authentication
  - rely on existing and proven mechanisms; do not re-invent the wheel
  - how to deal with authentication and identity in multi-layer environment?
  - use encrypted connections
    - confidentiality
    - avoid replay attach by using nonce, timestamp, seq. number. . .
  - Implement (re-use) CAPTCHA to avoid
    - brute-force attack
    - information theft

Architecture and design

- Authorization
  - adopt a proven and appropriate model
  - mediate all accesses (no backdoor)
- Use of cryptography
  - select existing and proven algorithms, protocols and implementations
  - particular attention to PRNG, key and confidential information storage

Architecture and design

- Integration of third-party components
  - assess the level of security by all means (tests, sandboxing, forums. . .)
  - validate dependencies, incl. when performing update
- Deployment/operation
  - activate only necessary components
  - secure configuration and default values
  - disable admin interface and debugging, or make it local
  - operation isolated environments

Development

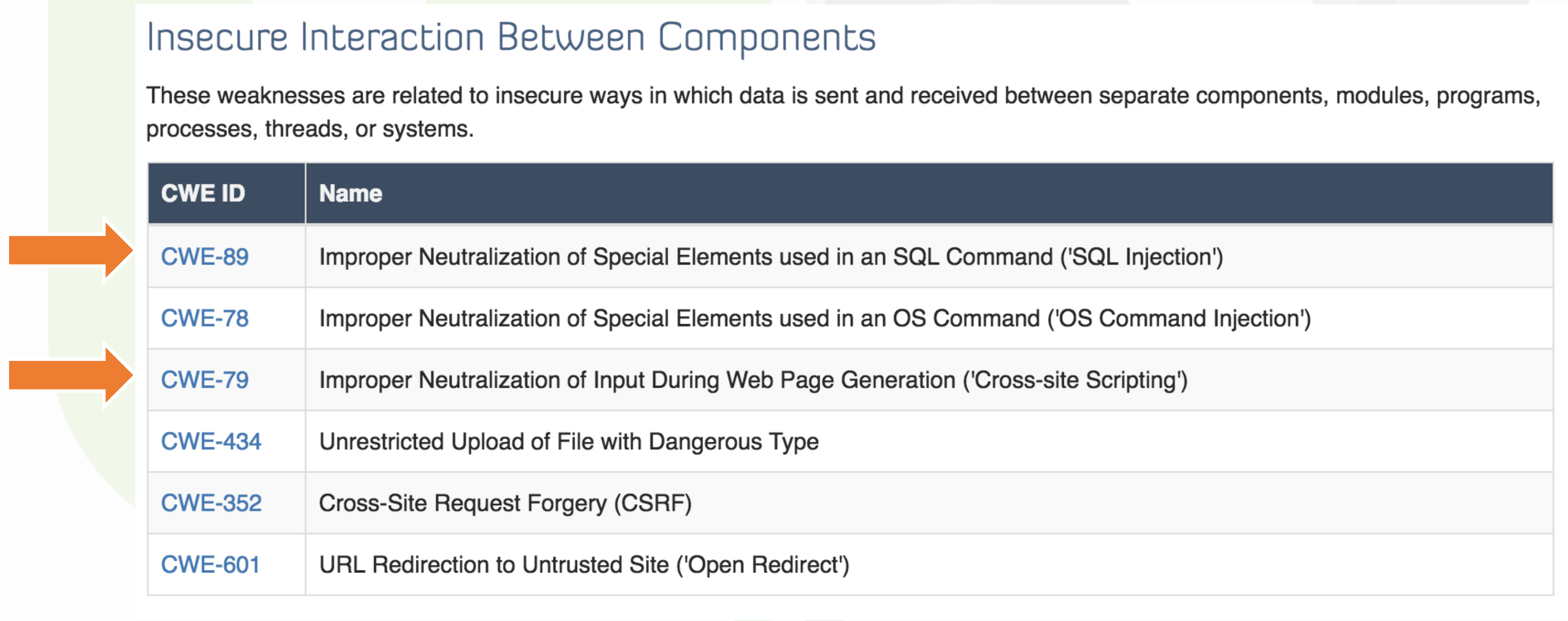
Top 10 Flaws: Do not …

1. Assume trust, rather than explicitly give it or award it
2. Use an authentication mechanism that can be bypassed or tampered with
3. Authorize without considering sufficient context
4. Confuse data and control instructions, and process control instructions from untrusted sources
5. Fail to validate data explicitly and comprehensively
6. Fail to use cryptography correctly
7. Fail to identify sensitive data and how to handle it
8. Ignore the users
9. Integrate external components without considering their attack surface
10. Rigidly constrain future changes to objects and actors

Source: “AVOIDING THE TOP 10 SOFTWARE SECURITY DESIGN FLAWS”, IEEE Center for Secure Design, https://computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf

Development

- SANS Top 25 Most Dangerous Software Errors 2017
  - https://www.sans.org/top25-software-errors

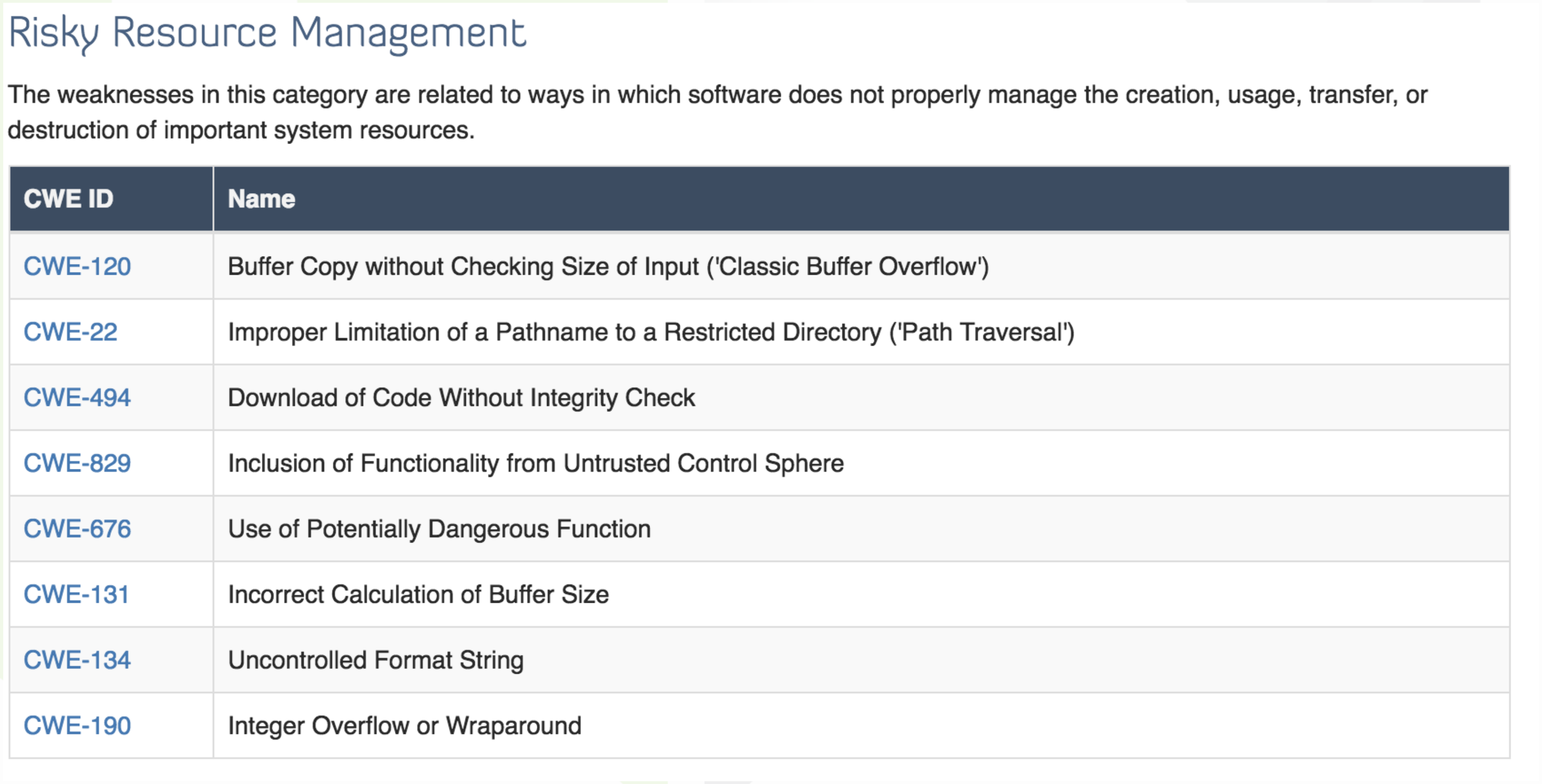


Insecure Interaction Between Components

These weaknesses are related to insecure ways in which data is sent and received between separate components, modules, programs, processes, threads, or systems.

CWE ID	Name
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-434	Unrestricted Upload of File with Dangerous Type
CWE-352	Cross-Site Request Forgery (CSRF)
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')

Development

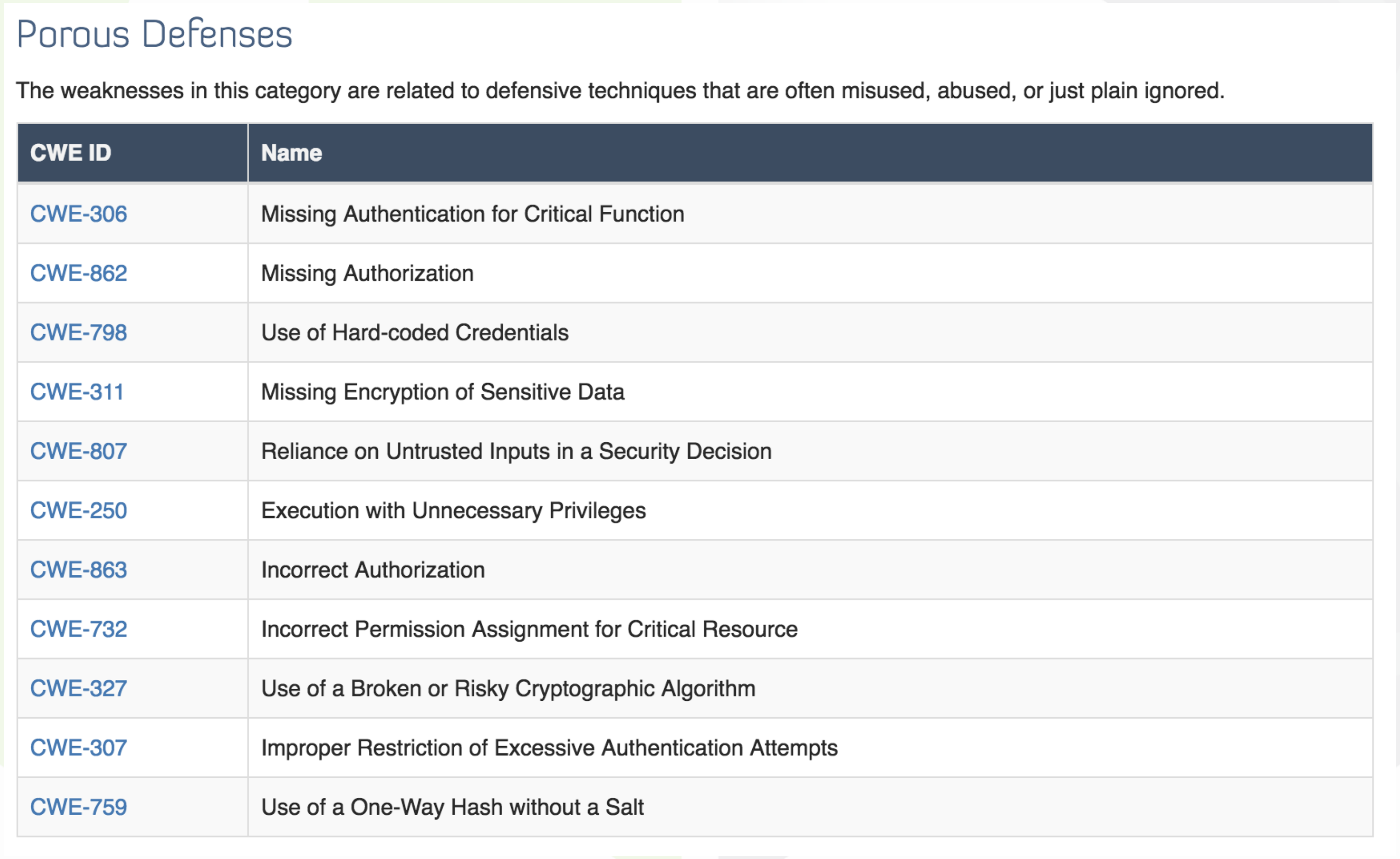


Risky Resource Management

The weaknesses in this category are related to ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources.

CWE ID	Name
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-494	Download of Code Without Integrity Check
CWE-829	Inclusion of Functionality from Untrusted Control Sphere
CWE-676	Use of Potentially Dangerous Function
CWE-131	Incorrect Calculation of Buffer Size
CWE-134	Uncontrolled Format String
CWE-190	Integer Overflow or Wraparound

Development



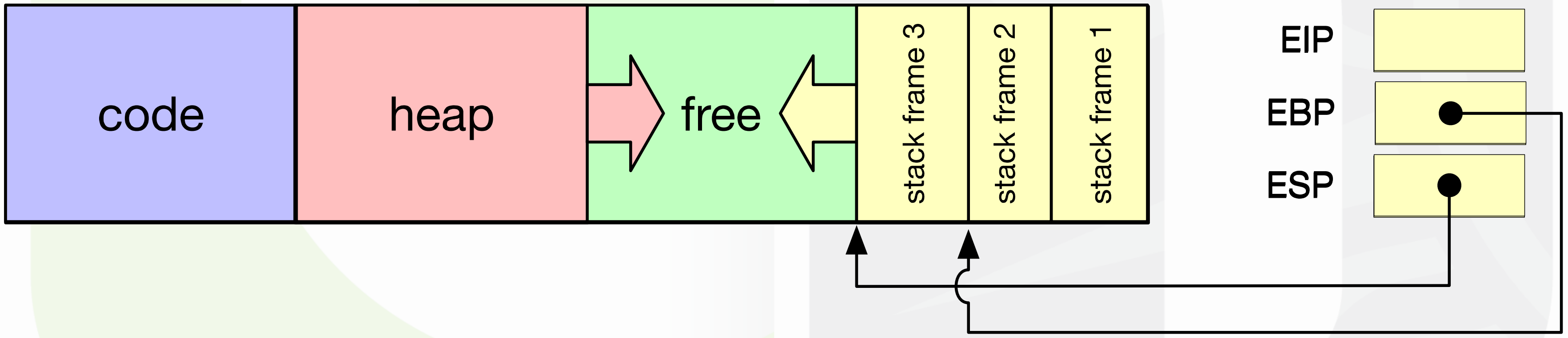
Porous Defenses

The weaknesses in this category are related to defensive techniques that are often misused, abused, or just plain ignored.

CWE ID	Name
CWE-306	Missing Authentication for Critical Function
CWE-862	Missing Authorization
CWE-798	Use of Hard-coded Credentials
CWE-311	Missing Encryption of Sensitive Data
CWE-807	Reliance on Untrusted Inputs in a Security Decision
CWE-250	Execution with Unnecessary Privileges
CWE-863	Incorrect Authorization
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-307	Improper Restriction of Excessive Authentication Attempts
CWE-759	Use of a One-Way Hash without a Salt

Buffer overflow

- IA32/64 Memory and registers (rxx for IA64)



Function call and stack

```
int main(int argc, char** argv) {
    myFunction3();
    return (EXIT_SUCCESS);
}
```

```
Dump of assembler code for function main:
0x08048414 <main+0>:    push   %ebp
0x08048415 <main+1>:    mov    %esp,%ebp
0x08048417 <main+3>:    call   0x8048423 <myFunction3>
0x0804841c <main+8>:    mov    $0x0,%eax
0x08048421 <main+13>:   pop    %ebp
0x08048422 <main+14>:   ret
End of assembler dump.
```

Function call and stack

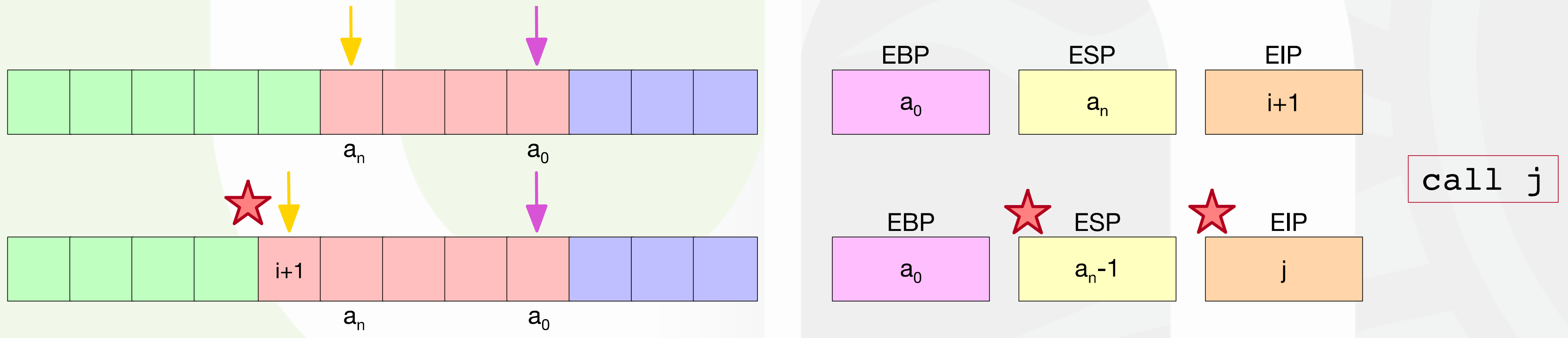
```
int myFunction3() {
  char buffer[30];
  gets(buffer);
  printf("%s\n", buffer);
}
```

```
Dump of assembler code for function myFunction3:
0x08048423 <myFunction3+0>:    push   %ebp
0x08048424 <myFunction3+1>:    mov    %esp,%ebp
0x08048426 <myFunction3+3>:    sub    $0x28,%esp
0x08048429 <myFunction3+6>:    lea    -0x1e(%ebp),%eax
0x0804842c <myFunction3+9>:    mov    %eax,(%esp)
0x0804842f <myFunction3+12>:   call   0x8048320 <gets@plt>
0x08048434 <myFunction3+17>:   lea    -0x1e(%ebp),%eax
0x08048437 <myFunction3+20>:   mov    %eax,(%esp)
0x0804843a <myFunction3+23>:   call   0x8048350 <puts@plt>
0x0804843f <myFunction3+28>:   leave
0x08048440 <myFunction3+29>:   ret
End of assembler dump.
```

Call function

call <addr>	push EIP jump <addr>

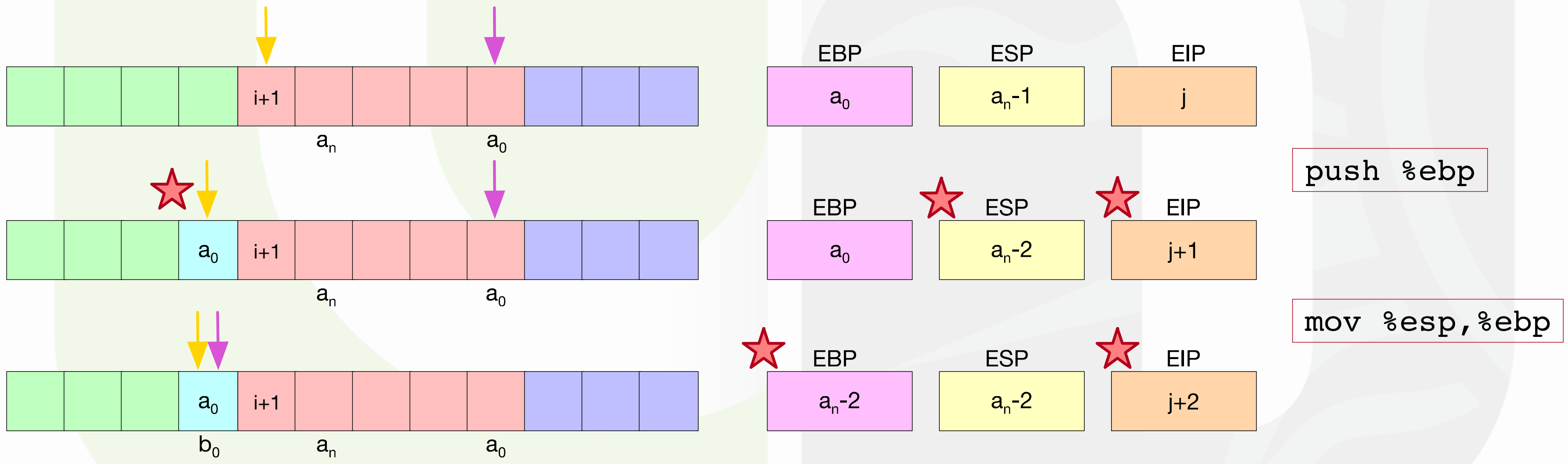
```
Dump of assembler code for function main:
0x08048414 <main+0>:     push    %ebp
0x08048415 <main+1>:     mov     %esp,%ebp
0x08048417 <main+3>:     call    0x8048423 <myFunction3>
0x0804841c <main+8>:     mov     $0x0,%eax
0x08048421 <main+13>:    pop     %ebp
0x08048422 <main+14>:    ret
End of assembler dump.
```



Increment is in arbitrary unit, not in Bytes

Save context

```
0x08048423 <myFunction3+0>:     push   %ebp
0x08048424 <myFunction3+1>:     mov    %esp,%ebp
0x08048426 <myFunction3+3>:     sub    $0x28,%esp
0x08048429 <myFunction3+6>:     lea    -0x1e(%ebp),%eax
0x0804842c <myFunction3+9>:     mov    %eax,(%esp)
0x0804842f <myFunction3+12>:    call   0x8048320 <gets@plt>
0x08048434 <myFunction3+17>:    lea    -0x1e(%ebp),%eax
0x08048437 <myFunction3+20>:    mov    %eax,(%esp)
0x0804843a <myFunction3+23>:    call   0x8048350 <puts@plt>
0x0804843f <myFunction3+28>:    leave
0x08048440 <myFunction3+29>:    ret
End of assembler dump.
```

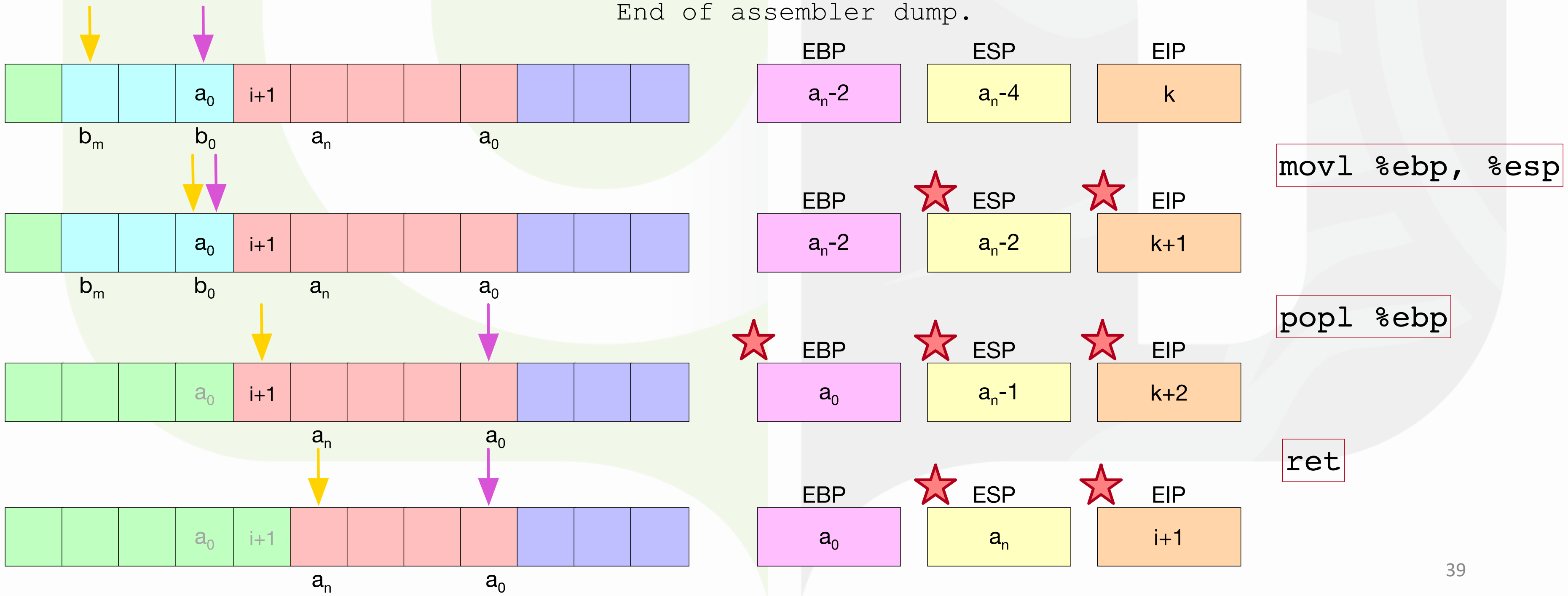


Increment is in arbitrary unit, not in Bytes

Return from function

leave	movl %ebp, %esp popl %ebp
ret	pop EIP

```
0x08048423 <myFunction3+0>:     push   %ebp
0x08048424 <myFunction3+1>:     mov    %esp,%ebp
0x08048426 <myFunction3+3>:     sub    $0x28,%esp
0x08048429 <myFunction3+6>:     lea    -0x1e(%ebp),%eax
0x0804842c <myFunction3+9>:     mov    %eax,(%esp)
0x0804842f <myFunction3+12>:    call   0x8048320 <gets@plt>
0x08048434 <myFunction3+17>:    lea    -0x1e(%ebp),%eax
0x08048437 <myFunction3+20>:    mov    %eax,(%esp)
0x0804843a <myFunction3+23>:    call   0x8048350 <puts@plt>
0x0804843f <myFunction3+28>:    leave
0x08048440 <myFunction3+29>:    ret
End of assembler dump.
```



Function call and stack

```
(gdb) break *0x08048423
Breakpoint 2 at 0x8048423: file overflow2.c, line 19.
(gdb) run
Starting program: /home/jnc/shellhack/overflow2
Breakpoint 2, myFunction3 () at overflow2.c:19
19        int myFunction3() {
(gdb) x $esp
0xbffff7c4:     0x0804841c
(gdb) x $ebp
0xbffff7c8:     0xbffff828
(gdb) si      (push   %ebp)
0x08048424      19        int myFunction3() {
(gdb) x $esp
0xbffff7c0:     0xbffff7c8
(gdb) x $ebp
0xbffff7c8:     0xbffff828
(gdb) si      (mov    %esp,%ebp)
0x08048426      19        int myFunction3() {
(gdb) x $esp
0xbffff7c0:     0xbffff7c8
(gdb) x $ebp
0xbffff7c0:     0xbffff7c8
```

Function call and stack

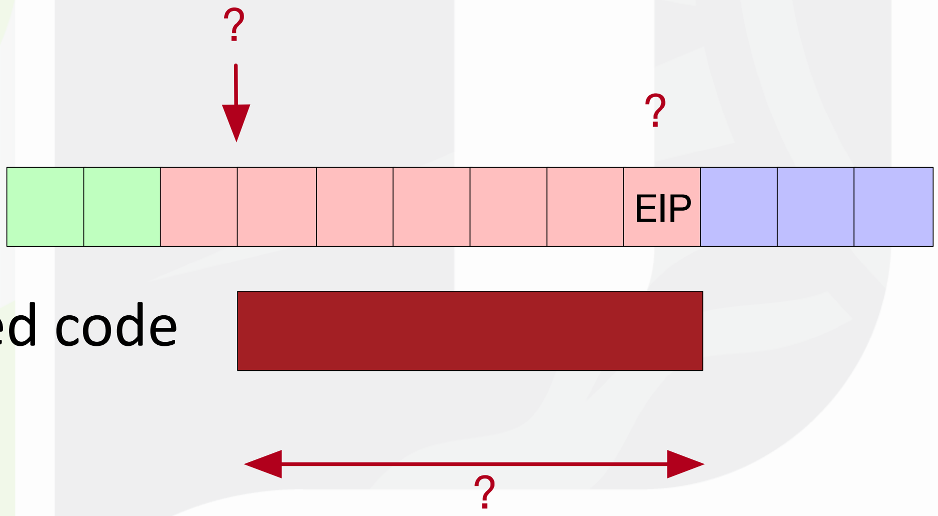
```
(gdb) si         (sub    $0x28,%esp)
(gdb) x $esp
0xbffff798:     0xbffff7a8
(gdb) x $ebp
0xbffff7c0:     0xbffff7c8
(gdb) x/20x $esp
0xbffff798:     0xbffff7a8      0x080482fc      0x00b5034e      0x080496b8
0xbffff7a8:     0xbffff7c8      0x080484c9      0x00b496d0      0x08048360
0xbffff7b8:     0x080484bb      0x00ccdff4      0xbffff7c8      0x0804841c
0xbffff7c8:     0xbffff828      0x00b746e5      0x00000001      0xbffff854
0xbffff7d8:     0xbffff85c      0xb7fea2d8      0x00000001      0x00000001
(gdb) s
AAAAAAAAABBBBBBBBBCCCCCCCCC
(gdb) x/20x $esp
0xbffff798:     0xbffff7a2      0x080482fc      0x4141034e      0x41414141
0xbffff7a8:     0x41414141      0x42424242      0x42424242      0x43434242
0xbffff7b8:     0x43434343      0x43434343      0xbffff700      0x0804841c
0xbffff7c8:     0xbffff828      0x00b746e5      0x00000001      0xbffff854
0xbffff7d8:     0xbffff85c      0xb7fea2d8      0x00000001      0x00000001
```

Buffer overflow

- Modify execution flow
  - overwrite RET address

```
> printf "AAAAAAAAAABBBBBBBBBB" | ./overflow2
AAAAAAAAAABBBBBBBBBB
> printf "AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDD\x17\x84\x04\x08" | ./overflow2
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDD#
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDD#
```

- Code injection
  - create shellcode
  - identify address of injected code
  - identify new RET address



Shellcode creation

- Assembly code
  - start from disassembled existing code
- Transform into binary code (opcode)
- Opcode must be as small as possible
- No 0x00 values
- How many instructions to perform fork/exec(“/bin/bash”)?

Shellcode creation

```
Dump of assembler code for function _exit:
0x0804f180 <_exit+0>:   mov    0x4(%esp),%ebx
0x0804f184 <_exit+4>:   mov    $0xfc,%eax
0x0804f189 <_exit+9>:   int    $0x80
0x0804f18b <_exit+11>:  mov    $0x1,%eax
0x0804f190 <_exit+16>:  int    $0x80
0x0804f192 <_exit+18>:  hlt
End of assembler dump.

[jnc@pelican shellhack]$ cat shellcode.asm
Section .text
   global _start

_start:
   mov ebx, 0
   mov eax, 1
   int 0x80
[jnc@pelican shellhack]$ nasm -f elf shellcode.asm
[jnc@pelican shellhack]$ ld -o shellcode shellcode.o
```

Shellcode creation

```
[jnc@pelican shellhack]$ objdump -d shellcode

shellcode:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    $0x0,%ebx
 8048065:       b8 01 00 00 00          mov    $0x1,%eax
 804806a:       cd 80                   int    $0x80

Shellcode  = `\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80'
```

Address identification

- shellcode address
  - guess from ESP
- RET address
  - issue
    - <shellcode><padding><RET>
    - determine padding length
  - different methods
    - ex: brute force
    - NOP sled

Format string bug

- Linked to C/C++ language (and those that rely upon tools in those languages)
- Format string used in many functions
  - ex: `printf("%0.2f", value);`
  - first param is the format string that defines display format and value placeholders
  - following params define values and variables to read from
- Problem when user can manipulate first parameter
    - what if no format string?
    - what if no variable while the format string specifies placeholders?

Format string bug

```
#include <stdio.h>
main() {
  printf("%x %x %x %x\n");
}
```

```
Breakpoint 1, 0x080483d1 in main () at essai1.c:4
4          printf("%x %x %x %x\n");
Missing separate debuginfos, use: debuginfo-install glibc-2.9-3.i686
(gdb) x/20x $esp
0xbffff7c0:     0x080484a4      0x08048310      0xbffff828      0x00b746e5
0xbffff7d0:     0x00000001      0xbffff854      0xbffff85c      0xb7fea2d8
0xbffff7e0:     0x00000001      0x00000001      0x00000000      0x0804822c
0xbffff7f0:     0x00ccdff4      0x080483f0      0x08048310      0xbffff828
0xbffff800:     0xf5741ab5      0x6416efcb      0x00000000      0x00000000

(gdb) c
Continuing.
8048310 bffff828 b746e5 1
```

Format string bug

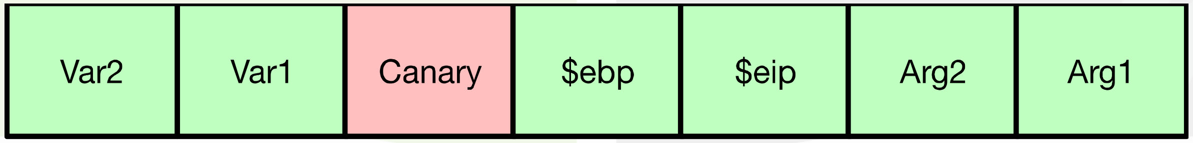
- Some format specifiers
  - %s expects a pointer to a NTS
  - %n expects a pointer to an int; the number of chars printed so far is written to the specified address
- Risks
  - information disclosure
  - execution flow modification
  - code injection

Other attacks

- Return-to-libc
  - idea: instead of injecting code, use existing code (ex. from libc)
  - modify EIP to point to existing code
  - modify local variables to use as parameters
- Return Oriented Programming
  - idea: use snippets of code in compiled program, and jump from snippet to snippet using RET address
  - each snippet ends with RET
  - change EIP
  - where to find snippets?
    - disassemble code, look for RET, check preceding instructions
  - Blind ROP: successful ROP despite ASLR and canaries (2015)

Counter-measures

- Memory protection
  - make heap and stack non-executable
- Canary
  - add a marker before the area where registers are saved
  - check marker before using RET
  - if RET has been overwritten, so is the marker
  - use random value for marker



Counter-measures

- ASLR – Address Space Layout Randomization
  - idea: place libraries and other elements (stack. . .] at addresses that change randomly between executions
  - implemented in most current OSes
  - needs sufficient entropy: on a 32 bits system, only 16 to 20 bits for randomization ⇒ vulnerable to brute force attack
  - risk of information leakage, for instance using a format string attack to reveal addresses
  - does not apply to compiled executable

Code injection

- Principle
  - happens when data entered by the user are used to build a command without first sanitizing it
    - best known: SQL Injection
    - but also: OS command injection, LDAP injection, XPATH injection, XML injection…

SQL injection

- Vulnerability: SQL request built from raw user data

```
select * from users where username = 'jnc' and password = 'mysecret’
```

```
String queryString = "select * from users "
              + "where username = '" + username + "' "
              + "and password = '" + password + "'";
       stmt = connection.createStatement();
       rs = stmt.executeQuery(queryString);
       if (rs.next()) {
            return true;
       } else {
            return false;
       }
```

SQL injection

- Normal case
  - variables:
    - username: `jnc`
    - password: `mysecret`

```
select * from users where username = 'jnc' and password = 'mysecret'
```

- Condition cancellation
  - variables:
    - username: `jnc`
    - password: `dunno’ or ’1’=’1`

```
select * from users where username = 'jnc' and
password = 'dunno' or '1'='1'
```

  - condition is always true
  - leads to information leakage, incorrect authentication, identity spoofing. . .

SQL injection

- where clause truncation
- variables:
  - username: `dunno’ or ’1’=’1’--`
  - password: `dunno`

```
select * from users where username = 'dunno' or
'1'='1'--' and password = 'dunno'
```

- cancels remaining conditions
- leads to information leakage, incorrect authentication, data integrity. . .

SQL injection

- Multiple requests
- variables:
  - username: `jnc`
  - password: `dunno’ ; delete * from invoices where ’1’ = ’1`

```
select * from users where username = 'jnc' and
password = 'dunno' ; delete * from invoices where '1' = '1'
```

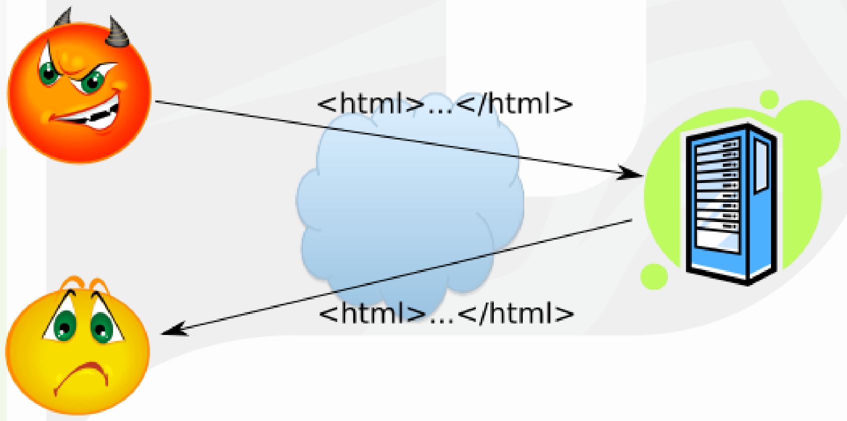
- request(s) insertion
- leads to data confidentiality and integrity risks

SQL injection

- Vulnerabilities
  - no validation of input data (sanitization)
  - query is dynamically built
  - level of risk linked to privilege of user that submits the query
  - error message can disclose useful information to attacker
- Counter measures
  - use prepared statements and variables linkage
  - validate input data
  - avoid verbose error messages
  - log accesses (and analyze it)
  - least privilege/separation of duty

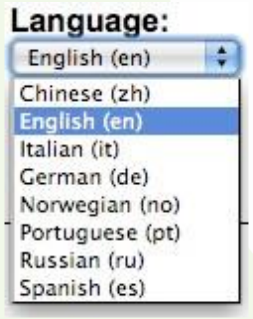
Cross-site scripting

- Data entered by user is used as html code to other users
  - ex: blogs, forums
    1. `<script>alert("XSS!")</script>`
    2. `<img src="javascript:alert ('XSS')">`
    3. `<body onload="javascript:alert('XSS')">`
- Counter-measures
  - input data validation
  - non-trivial task



Client-side validation

- client (browser, app. . .) is under user's control
- HTML select control (list, checkbox...)
  - used to limit the value space



```
<strong>Language:</strong>
<select name="language">
<option value="zh">Chinese (zh)</option>
<option selected value="en">English (en)</option>
<option value="it">Italian (it)</option>
<option value="de">German (de)</option
<option value="no">Norwegian (no)</option>
<option value="pt">Portuguese (pt)</option>
<option value="ru">Russian (ru)</option>
<option value="es">Spanish (es)</option>
</select>
```

```
http://localhost:8080/AntiSamyDemoWebApp/?profile=%3Cb%3EHello%3C%2Fb%3E%0D
%0A%3Cscript%3Ealert%28%27Hello%27+%2B+document.cookie%29%3C%2Fscript%3E&po
licy=NO+POLICY&language=de
```

Client-side validation

- Use of JavaScript, hidden fields. . .
- Easily circumvented
  - save html locally, edit and run local copy
  - deactivate JavaScript
  - use a proxy
- Solution
  - client-side validation for user-friendliness
  - server-side validation for security

XXE – XML eXternal Entity

- Occurs when an XML document includes a reference to an external *entity* element

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <!DOCTYPE foo [
3.      <!ELEMENT foo ANY >
4.      <!ENTITY xxe SYSTEM "file:///etc/passwd>" >]>
5.  <foo>&xxe;</foo>
```

- Counter-measures
  - Disable external entity processing
  - Strong validation

Directory traversal

- Unauthorized access to filesystem or other resources
  - incl. application components
- Ex.
  - http://mywebsite.org/getReport.jsp?file=getReport.jsp
  - http://mywebsite.org/getReport.jsp?file=**../../../etc/passwd**
- Check patterns in URLs
- Counter-measures
  - limit access to webroot
    - sandbox, chroot, root jail
  - use ACL – Access Control List
  - validate input data

Session management

- Problem: HTTP is a stateless protocol, while there is a need to preserve a state across HTTP exchanges
  - ex: shopping card
- Common approach: store a session identifier that is passed from browser to server at each request
  - hidden form field
  - CGI or URL variables
  - cookies
    - persistent vs non-persistent
    - secure vs non-secure
    - must be as opaque as possible; encrypt any sensitive information
    - do not trust expiry date (can be modified by user)
- all these data can be easily manipulated by user!

Session management

- Session Hijacking
  - theft of session identifiers (sniffing, cookie theft. . .)
  - secure session identifiers
    - generate random or unforgeable value
    - expire and renew
    - cleanup when logout
    - use framework functionalities
- Navigation sequence
  - goal: check the sequence of pages visited by user
  - maintain a navigation history
    - preferably server-side (if client side, risk of user tampering)

Security-related HTTP headers

- HTTP traffic security can be improved with some headers
  - [Content-Security-Policy] defines acceptable content sources (mitigates XSS)
  - [X-XSS-Protection] enables the XSS filter of the browser
  - [HTTP Strict Transport Security (HSTS)] enforces https traffic
  - [X-Frame-Options] prevents iframes (mitigates clickjacking)
  - [Expect-CT] checks certificate validity with CT – Certificate Transparency
  - [X-Content-Type-Options] instructs the browser to strictly follow the advertised mime types (avoid MIME sniffing)
  - [Feature-Policy] defines acceptable use of browser features (geolocation, camera, micro. . .)

Methods and tools

Source code verification

- Compiler checks
- Quality metrics
- Static analysis
  - focus on obvious cases or bad practices
  - use while coding or as part of a validation process
  - need to be properly configured to avoid false positive/false negative (⇒ loss of trust)
- Peer review. Requires
  - good coding standards
  - list of vulnerabilities

Static analysis

- Can detect
  - calls to potentially unsafe functions
  - bound validation
  - typing errors (type confusion, also for pointers)
  - memory allocation
  - invalid sequences of operations
  - tainted data analysis
- ex: HP Fortify, CodeSonar, IBM Rational AppScan,
- http://findbugs.sourceforge.net/
- https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Testing

- Software testing vs Software Security Testing
  - adopt attacker perspective: try to break it!!
  - focus on what the system should not do instead of just testing expected behavior
  - bug detection requires trying exception cases, beyond normal behavior
  - functional testing: are (security) specs properly implemented?
    - ex: deactivate account after 3 unsuccessful login attempts
  - risk-based testing
    - testing process guided by threat model
    - look for common errors or vulnerabilities
    - rely on and improve knowledge base and skills
      - previous incident reports. . .

Testing

- Blackbox testing
  - consider the software system as a blackbox
  - can test
    - input data validation
    - code injection and buffer overflow
    - XSS
    - directory traversal
    - session management
    - authentication and access control mechanisms
  - fuzzing: automatically inject (not so) random data
  - server and application fingerprint
  - load testing
  - behavior analysis through data collection (input/output values, ex. cookies)

Penetration Testing (pentest)

- Goal: systematic testing of a system security, trying to identify and possibly exploit vulnerabilities
- often requires not to be detected
- use of tools
  - reconnaissance (network and port scanning, vulnerability scanning. . .)
  - exploitation: various available frameworks that integrate exploits and payloads for a large variety of vulnerabilities and systems
  - ex: nmap, metasploit, openvas, nessus, nikto. . .

Many tools exist

- A lot of tools are available to help you develop secure code
    - Use them, but check that they are reliable and well-supported
- Ex: OWASP tools
    - OWASP Testing Framework
    - OWASP Zed Attack Proxy
    - OWASP Web Testing Environment
    - OWASP Enterprise Security API

Conclusion

- Security has to be woven into the software product
- Integrate security concerns from the start, and throughout SDLC
- Think like an attacker
- Define, adapt, improve, apply best practices for all SDLC steps
- Organize a review of source code
- Define test scenarios based on risk analysis
- Develop the in-house competence