PV286 - Secure coding principles and practices

Secure coding introduction + language level vulnerabilities: Buffer overflow, type overflow, strings

Łukasz Chmielewski ***chmiel@fi.muni.cz*** **(based on the lecture by P. Svenda)**
(email me with your questions/feedback)
Centre for Research on Cryptography and Security, Masaryk University
Consultation hours: Friday 9.30-11.00 in A406 (but email me before).

CRoCS
Centre for Research on Cryptography and Security

This Lecture

- Course trivia: PV286+PA193
- Short Project Presentation (by Jan Kvapil)
- The lecture itself. If we do not finish, then please check:
  - https://is.muni.cz/auth/el/fi/jaro2022/PA193/um/video/PA193_01_BufferOverflowStrings_2022.video5
- Materials:
  - https://is.muni.cz/auth/el/fi/jaro2024/PV286/um/

COURSE TRIVIA:

PV286+PA193_00_COURSE_ORGANISATION_2024

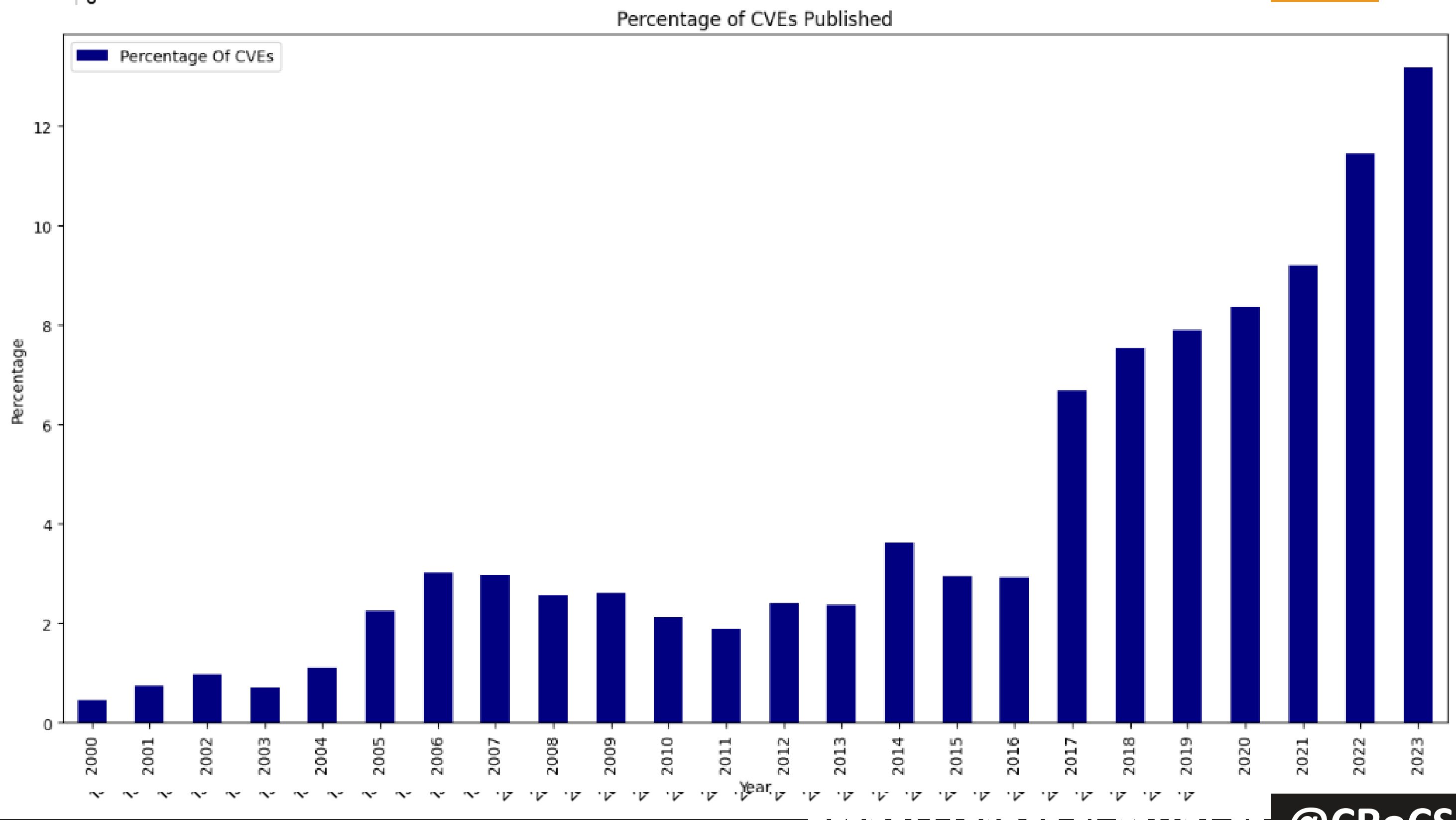
Last 20 Scored Vulnerability IDs & Summaries — CVSS Severity

CVE-2020-7558 - A CWE-787 Out-of-bounds Write vulnerability exists in IGSS Definition (Def.exe) version 14.0.0.20247 that could cause Remote Code Execution when malicious CGF (Configuration Group File) file is imported to IGSS Definition.
Published: November 19, 2020; 5:15:14 PM -0500
V3.1: 7.8 HIGH
V2.0: 6.8 MEDIUM

CVE-2020-13877 - SQL Injection issues in various ASPX pages of ResourceXpress Meeting Monitor 4.9 could lead to remote code execution and information disclosure.
Published: November 12, 2020; 4:15:10 PM -0500
V3.1: 9.8 CRITICAL
V2.0: 7.5 HIGH

CVE-2020-12353 - Improp
version 3.6.2 may allow ar
access.
Published: November 12,
MEDIUM

Total Matches By Year
18,000



.nist.gov/

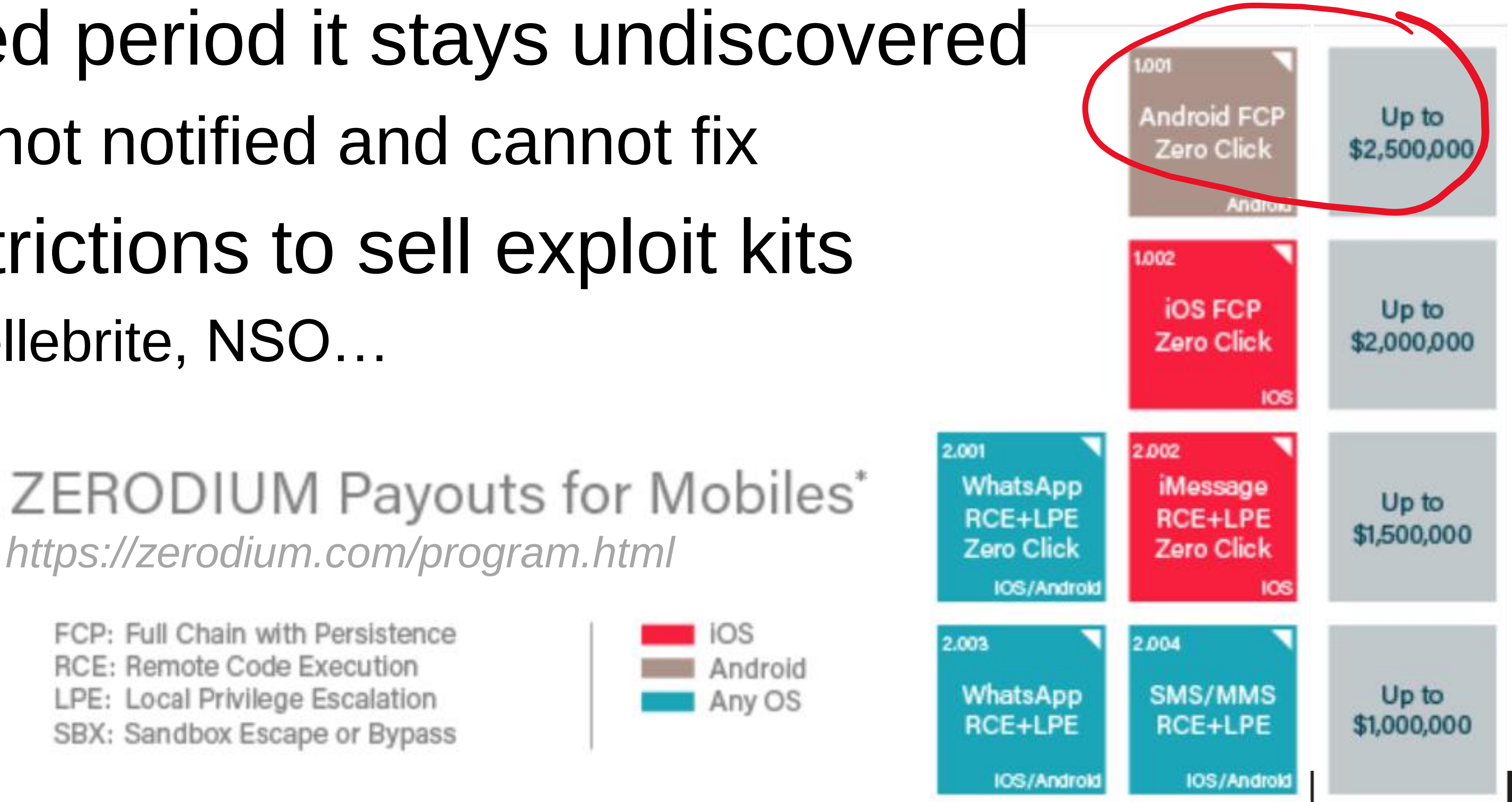
Problem?

What is the cost of insecure software

- Increased risk and failures due to generally increased usage of computers
- Fixing bug in released version is more expensive
  - Testing, announcements…
- Liability laws
  - Need to notify, settlements, GDPR...
- Reputation loss
  - (unfortunately, does not seem to be at the moment)
- Cost of defense is decreasing
  - better training (like this course ☺), automated tools, development methods, new langs…
  - but the complexity of software is also increasing

There is HUGE market for (undisclosed) vulnerabilities

- Up to millions of dollars for single undisclosed exploit
- Payed over defined period it stays undiscovered
  - Product vendor is not notified and cannot fix
- Ethics: export restrictions to sell exploit kits
  - But HackingTeam, Cellebrite, NSO…



https://zerodium.com/program.html

What software security means?

- Use of generic good development and security practices
  - Education, testing, defence in depth, code review…
  - Safety (random errors CRC good enough) vs. security (intentional attacker recomputing CRC after malicious change)
  - Security is process, not product (Secure Development Lifecycle)
- Have systematic deployment, maintenance and mitigation of issues (including the security relevant)
  - Monitor, triage, fix, update process, detection of issues in 3rd party libs…
- Usability - easy to use right, hard to misuse
  - Hard for developers to misuse or misconfigure (API security…), hard for end-users to make a mistake
  - If misuse, then limit its impact, secure defaults…
- Automated and manual review and testing
  - Continuous integration, pentesting, security code review
- Language-specific issues and procedures, corresponding tooling and automation
  - Buffer overflow (C/C++), code injection (Java)…
- Use of secure cryptographic primitives
  - Cryptographic libraries, random numbers, password handling, secure channels, key distribution…

Icons made by geotatah, eucalypt, freepik from www.flaticon.com

Defensive programming

- Term coined by Kernighan and Plauger, 1981
  - *"writing the program so it can cope with small disasters"*
  - talked about in introductory programming courses
- Practice of coding with the mind-set that errors are inevitable, and something will always go wrong
  - prepare program for unexpected behavior and inputs
  - prepare program for easier testing and bug diagnostics
- Defensive programming targets mainly unintentional errors (not intentional attacks)
  - But increasingly given security connotation

WHERE TO LEARN ABOUT BUGS AND RESULTING VULNERABILITIES?

Attacker goals and related vulnerabilities

- Bug is unintended and unwanted behavior which attacker can use to:

1. Steal some data (keys in memory, content of files…)
2. Bypass some protection (access rights, authentication, hijack session)
3. Execute malicious code (custom payload, ROP…)
4. Cause denial of service (resource exhaustion, infinite loop, regex)
5. …

- The real attack (exploit) often combines multiple steps
  - E.g., DoS to deplete memory resulting in failed dynamic allocation, then write to null pointer, then execute malicious payload

Where to find relevant bug patterns and info

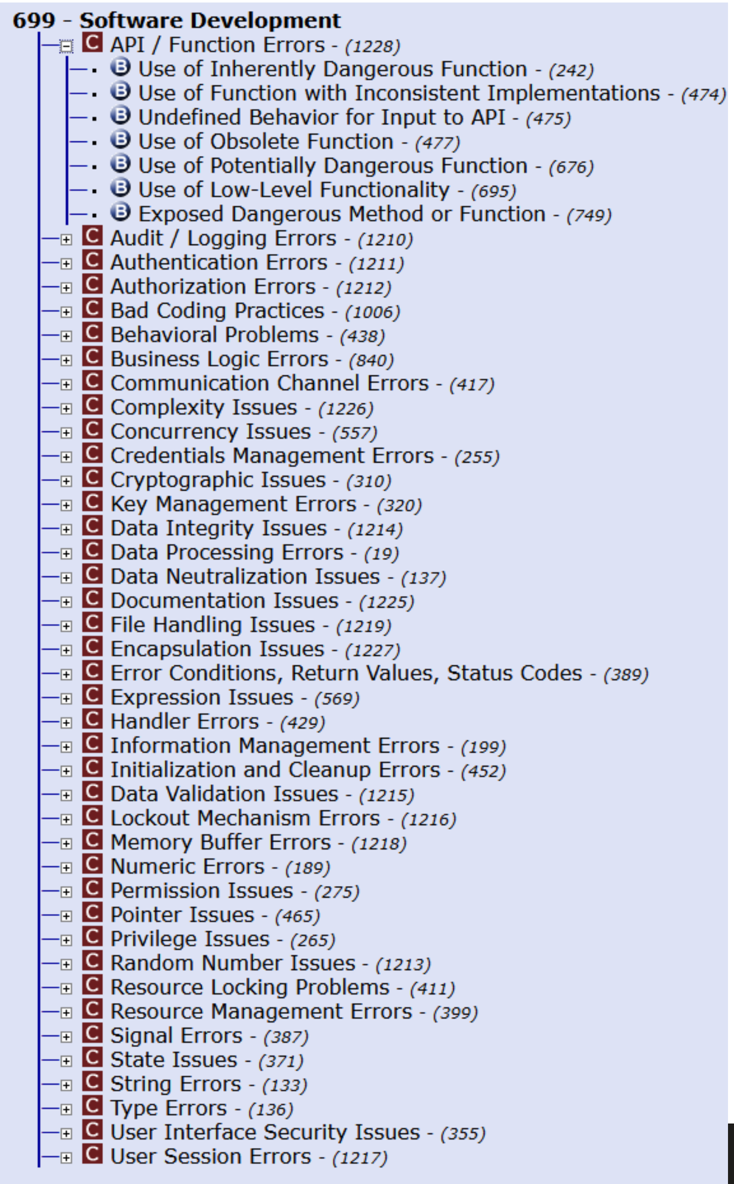
- Taxonomies of vulnerabilities (systematic)
  - Common Weakness Enumeration (CWE) https://cwe.mitre.org/
  - Wikipedia (https://en.wikipedia.org/wiki/Memory_safety ...)
- List of real vulnerabilities detected and reported (complex real-world examples)
  - Common Vulnerabilities and Exposures (CVE) https://cve.mitre.org/
- Lists of frequent bugs (prioritization)
  - The CWE Top 25 https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html
  - OWASP TOP10 https://owasp.org/www-project-top-ten/
  - HackerOne TOP 10 https://www.hackerone.com/top-10-vulnerabilities
  - Veracode TOP 10 by language https://info.veracode.com/state-of-software-security-volume-11-flaw-frequency-by-language-infosheet-resource.html
  - Significant differences between usage domains (web vs. embedded devices)
- Bug patterns searched for by specific tool (understanding bugs & tool used)
  - E.g., FindSecurityBugs (Java): https://find-sec-bugs.github.io/bugs.htm

CROCS

Common Weakness Enumeration (CWE)

- Taxonomy of vulnerabilities https://cwe.mitre.org/
- List of vulnerability categories, sub-categories, examples and mitigation
  - Baseline for vulnerability identification, mitigation and prevention
  - Itself is great study material including examples
- Example CWE-124 Buffer Underwrite
  - https://cwe.mitre.org/data/definitions/124.html

```
int main() {
  // …
  strncpy(destBuf, &srcBuf[find(srcBuf, ch)], 1024);
}
```



699 - Software Development
- API / Function Errors - (1228)
  - Use of Inherently Dangerous Function - (242)
  - Use of Function with Inconsistent Implementations - (474)
  - Undefined Behavior for Input to API - (475)
  - Use of Obsolete Function - (477)
  - Use of Potentially Dangerous Function - (676)
  - Use of Low-Level Functionality - (695)
  - Exposed Dangerous Method or Function - (749)
- Audit / Logging Errors - (1210)
- Authentication Errors - (1211)
- Authorization Errors - (1212)
- Bad Coding Practices - (1006)
- Behavioral Problems - (438)
- Business Logic Errors - (840)
- Communication Channel Errors - (417)
- Complexity Issues - (1226)
- Concurrency Issues - (557)
- Credentials Management Errors - (255)
- Cryptographic Issues - (310)
- Key Management Errors - (320)
- Data Integrity Issues - (1214)
- Data Processing Errors - (19)
- Data Neutralization Issues - (137)
- Documentation Issues - (1225)
- File Handling Issues - (1219)
- Encapsulation Issues - (1227)
- Error Conditions, Return Values, Status Codes - (389)
- Expression Issues - (569)
- Handler Errors - (429)
- Information Management Errors - (199)
- Initialization and Cleanup Errors - (452)
- Data Validation Issues - (1215)
- Lockout Mechanism Errors - (1216)
- Memory Buffer Errors - (1218)
- Numeric Errors - (189)
- Permission Issues - (275)
- Pointer Issues - (465)
- Privilege Issues - (265)
- Random Number Issues - (1213)
- Resource Locking Problems - (411)
- Resource Management Errors - (399)
- Signal Errors - (387)
- State Issues - (371)
- String Errors - (133)
- Type Errors - (136)
- User Interface Security Issues - (355)
- User Session Errors - (1217)

https://cwe.mitre.org/data/definitions/124.html

CWE-124: Buffer Underwrite ('Buffer Underflow')

Weakness ID: 124
Abstraction: Base
Structure: Simple

Status: Incomplete

Presentation Filter: Complete

Description

Extended Description

This typically occurs when a pointer or its index is decremented to a position before the buffer, when pointer arithmetic results in a position before the beginning of the valid memory location, or when a negative index is used.

Alternate Terms

buffer underrun: Some prominent vendors and researchers use the term "buffer underrun". "Buffer underflow" is more commonly used, although both terms are also sometimes used to describe a buffer under-read (CWE-127).

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOf and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that the user may want to explore.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	*Type*	*ID*	*Name*
ChildOf	B	787	Out-of-bounds Write
ChildOf	B	786	Access of Memory Location Before Start of Buffer
CanFollow	B	839	Numeric Range Comparison Without Minimum Check

Relevant to the view "Software Development" (CWE-699)

Nature	*Type*	*ID*	*Name*
MemberOf	C	1218	Memory Buffer Errors

Modes Of Introduction

Applicable Platforms

The listings below show possible areas for which the given weakness could appear. These may be for specific named Languages, Operating Systems, Architectures, Paradigms, Technologies, or a class of such platforms. The platform is listed along with how frequently the given weakness appears for that instance.

Languages

C *(Undetermined Prevalence)*

C++ *(Undetermined Prevalence)*

Common Consequences

The table below specifies different individual consequences associated with the weakness. The Scope identifies the application security area that is violated, while the Impact describes the negative technical impact that arises if an adversary succeeds in exploiting this weakness. The Likelihood provides information about how likely the specific consequence is expected to be seen relative to the other consequences in the list. For example, there may be high likelihood that a weakness will be exploited to achieve a certain impact, but a low likelihood that it will be exploited to achieve a different impact.

Common Consequences

The table below specifies different individual consequences associated with the weakness. The Scope identifies the application security area that is violated, while the Impact describes the negative technical impact that arises if an adversary succeeds in exploiting this weakness. The Likelihood provides information about how likely the specific consequence is expected to be seen relative to the other consequences in the list. For example, there may be high likelihood that a weakness will be exploited to achieve a certain impact, but a low likelihood that it will be exploited to achieve a different impact.

Scope	Impact	Likelihood
Integrity Availability	**Technical Impact:** *Modify Memory; DoS: Crash, Exit, or Restart* Out of bounds memory access will very likely result in the corruption of relevant memory, and perhaps instructions, possibly leading to a crash.	
Integrity Confidentiality Availability Access Control Other	**Technical Impact:** *Execute Unauthorized Code or Commands; Modify Memory; Bypass Protection Mechanism; Other* If the corrupted memory can be effectively controlled, it may be possible to execute arbitrary code. If the corrupted memory is data rather than instructions, the system will continue to function with improper changes, possibly in violation of an implicit or explicit policy. The consequences would only be limited by how the affected data is used, such as an adjacent memory location that is used to specify whether the user has special privileges.	
Access Control Other	**Technical Impact:** *Bypass Protection Mechanism; Other* When the consequence is arbitrary code execution, this can often be used to subvert any other security service.	

Likelihood Of Exploit

Demonstrative Examples

Example 2

The following is an example of code that may result in a buffer underwrite, if find() returns a negative value to indicate that ch is not found in srcBuf:

Example Language: **C** *(bad code)*

```
int main() {
  ...
  strncpy(destBuf, &srcBuf[find(srcBuf, ch)], 1024);
  ...
}
```

Observed Examples

Reference	Description
CVE-2002-2227	Unchecked length of SSLv2 challenge value leads to buffer underflow.
CVE-2007-4580	Buffer underflow from a small size value with a large buffer (length parameter inconsistency, CWE-130)
CVE-2007-1584	Buffer underflow from an all-whitespace string, which causes a counter to be decremented before the buffer while looking for a non-whitespace character.
CVE-2007-0886	Buffer underflow resultant from encoded data that triggers an integer overflow.
CVE-2006-6171	Product sets an incorrect buffer size limit, leading to "off-by-two" buffer underflow.
CVE-2006-4024	Negative value is used in a memcpy() operation, leading to buffer underflow.
CVE-2004-2620	Buffer underflow due to mishandled special characters

Potential Mitigations

Requirements specification: The choice could be made to use a language that is not susceptible to these issues.

Phase: Implementation

Sanity checks should be performed on all calculated values used as index or for pointer arithmetic.

Frequent bugs – worth of prioritization (CWE/CVE)

https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53
[18]	CWE-522	Insufficiently Protected Credentials	5.49
[19]	CWE-611	Improper Restriction of XML External Entity Reference	5.33
[20]	CWE-798	Use of Hard-coded Credentials	5.19
[21]	CWE-502	Deserialization of Untrusted Data	4.93
[22]	CWE-269	Improper Privilege Management	4.87
[23]	CWE-400	Uncontrolled Resource Consumption	4.14
[24]	CWE-306	Missing Authentication for Critical Function	3.85
[25]	CWE-862	Missing Authorization	3.77

- Score by presence in real vulnerabilities
  - Common Vulnerabilities and Exposures (CVE)

Frequent bugs – worth of prioritization (web)

Top 10 Web Application Security Risks OWASP https://owasp.org/www-project-top-ten/

1. **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
2. **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
3. **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
4. **XML External Entities (XXE)**. Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
5. **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
6. **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
7. **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
8. **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
9. **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
10. **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

- Be aware:
  - Differences between software domains (web, OS kernel, libraries…)
  - Detection bias – bugs we can more easily detect seem to be more frequent

Example: Injection (1. OWASP TOP 10, 3. CWE Top 25)

https://owasp.org/www-project-top-ten/2017/A1_2017-Injection

1. **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

- Goal: Return records from DB for the provided customer ID (`custID`)

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id")
+ "'";
```

- User/attacker will provide customer ID as follows:
  - `http://example.com/app/accountView?id=' or '1'='1`
- Resulting SQL command after expansion (executed by database engine)
  - `SELECT * FROM accounts WHERE custID='' or '1'='1'`
- Mitigation
  - Don't try to detect and fix injection by checking input arguments yourself!
  - Read about defenses, use dedicated secure API (e.g., PreparedStatement in this case)
  - https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

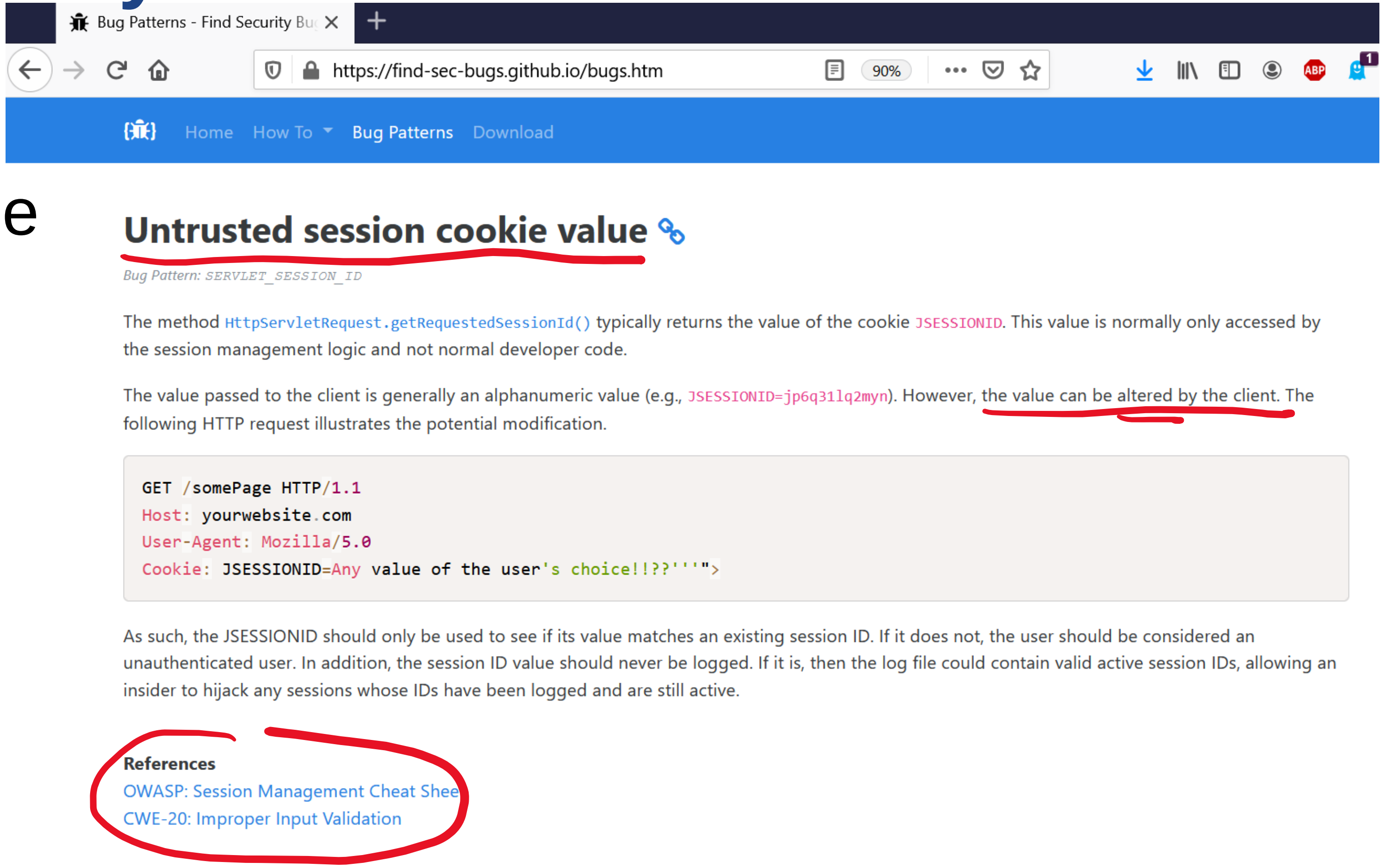
CWE flaw types by language

https://info.veracode.com/state-of-software-security-volume-11-flaw-frequency-by-language-infosheet-resource.html

	.Net	C++	Java	JavaScript	PHP	Python
1	Information Leakage 62.8%	Error Handling 66.5%	CRLF Injection 64.4%	Cross-Site Scripting (XSS) 31.5%	Cross-Site Scripting (XSS) 74.6%	Cryptographic Issues 35.0%
2	Code Quality 53.6%	Buffer Management Errors 46.8%	Code Quality 54.3%	Credentials Management 29.6%	Cryptographic Issues 71.6%	Cross-Site Scripting (XSS) 22.2%
3	Insufficient Input Validation 48.8%	Numeric Errors 45.8%	Information Leakage 51.9%	CRLF Injection 28.4%	Directory Traversal 64.6%	Directory Traversal 20.6%
4	Cryptographic Issues 45.9%	Directory Traversal 41.9%	Cryptographic Issues 43.3%	Insufficient Input Validation 25.7%	Information Leakage 63.3%	CRLF Injection 16.4%
5	Directory Traversal 35.4%	Cryptographic Issues 40.2%	Directory Traversal 30.4%	Information Leakage 22.7%	Untrusted Initialization 61.7%	Insufficient Input Validation 8.3%
6	CRLF Injection 25.3%	Code Quality 36.6%	Credentials Management 26.5%	Cryptographic Issues 20.9%	Code Injection 48.0%	Information Leakage 8.3%
7	Cross-Site Scripting (XSS) 24.0%	Buffer Overflow 35.3%	Cross-Site Scripting (XSS) 25.2%	Authentication Issues 14.9%	Encapsulation 48.0%	Server Configuration 8.1%
8	Credentials Management 19.9%	Race Conditions 30.2%	Insufficient Input Validation 25.2%	Directory Traversal 11.5%	Command or Argument Injection 45.4%	Credentials Management 7.2%
9	SQL Injection 12.7%	Potential Backdoor 25.0%	Encapsulation 18.1%	Code Quality 7.6%	Credentials Management 44.3%	Dangerous Functions 6.9%
10	Encapsulation 12.4%	Untrusted Initialization 22.4%	API Abuse 16.2%	Authorization Issues 4.0%	Code Quality 40.3%	Authorization Issues 6.8%

Bugs patterns searched by tools

- Bug description
- Example of vulnerable code
- References to other lists
  - – CWE, OWASP…



https://find-sec-bugs.github.io/bugs.htm

Digging deeper and learning more…

- Read top-level categories from CWE Software Development
  - Get broad overview https://cwe.mitre.org/data/definitions/699.html
- Read details about top vulnerabilities from OWASP or CWE list
  - Likely the most common ones
- Find, read about and test several vulnerabilities in detail
  - Which applies to your favorite language (e.g., Java)
  - And target domain (e.g., server database backend) in detail
  - Learn more about system by understanding all details
- Experiment with several automatic tools to detect such vulnerabilities
- Think like an attacker, have fun ☺

Vulnerability disclosure basics

- Bug, Vulnerability, Proof of Concept (PoC), Exploit
  - Bug = buffer overflow
  - Vulnerability = execution of malicious code
  - Proof of Concept = tool triggering buffer overflow and crashing program
  - Exploit = tool trigger buffer overflow, executing custom payload and creating root account on target machine
- Public disclosure, Uncoordinated public disclosure, Zero-day
- Responsible disclosure, disclosure period/deadline, bugbounty
- Whitehats, blackhats, red teams, blue teams

HOW TO PREVENT, DETECT AND MITIGATE CODE BUGS?

How to prevent, detect and mitigate code bugs?

1. Protection on the source code level
   - E.g., languages with/without implicit protection (containers/languages with array boundary checking)
   - E.g., input checking, sanitization, safe alternatives to vulnerable function like safe string manipulation
2. Protection by extensive testing (source code/binary/bytecode level)
   - E.g., automatic detection by static and dynamic checkers
   - E.g., code review, security testing
3. Protection by compiler (+ compiler flags)
   - E.g., runtime checks introduced by compiler (stack protection)
4. Protection by execution environment
   - E.g., DEP, ASLR, sandboxing, hardware isolation...
5. Protection by defense in depth
   - All above in systematic secure development lifecycle, multiple layers of defense

Microsoft's Secure Development Lifecycle (SDL)

Training	Requirements	Design	Implementation	Verification	Release	Response
	2. Establish Security Requirements	5. Establish Design Requirements	8. Use Approved Tools	11. Perform Dynamic Analysis	14. Create an Incident Response Plan	
1. Core Security Training	3. Create Quality Gates/Bug Bars	6. Perform Attack Surface Analysis/ Reduction	9. Deprecate Unsafe Functions	12. Perform Fuzz Testing	15. Conduct Final Security Review	Execute Incident Response Plan
	4. Perform Security and Privacy Risk Assessments	7. Use Threat Modeling	10. Perform Static Analysis	13. Conduct Attack Surface Review	16. Certify Release and Archive	

TOOLS!

EDU

METRICS

PREPARE FOR ISSUES

SIGN

https://www.microsoft.com/en-us/securityengineering/sdl/practices

Use secure-by default languages and libraries

- Ideally, language is already designed to be more secure
  - Partially true for newer languages like Go or Rust
  - But new systematic issues may be found later
- Libraries
  - Use functions from platform standard API (e.g., AndroidKeyStore provider)
  - Use libraries which are hard to be used incorrectly
    - E.g., Libsodium’s crypto_secretbox_easy() vs. OpenSSL vs. own custom code
  - Monitor used libraries/packages for new vulnerabilities (dependbot)
- Don’t design or implement own libraries especially not cryptographic
  - Developing own library code likely means repeating other’s mistakes
  - Cryptographic code is extremely difficult to code securely

Use of more secure versions of functions

- Consider language removing whole class of vulnerabilities
  - E.g., Rust to replace memory-related errors in C
- If language is fixed, then use more secure / hardened functions
  - E.g., Secure C library ISO/IEC 9899:2011
  - E.g., java.lang.Math precise arithmetic extensions
  - E.g., Smart pointers in C++

```
char *gets(
   char *buffer
);

char *gets_s(
   char *buffer,
   size_t sizeInCharacters
);
```

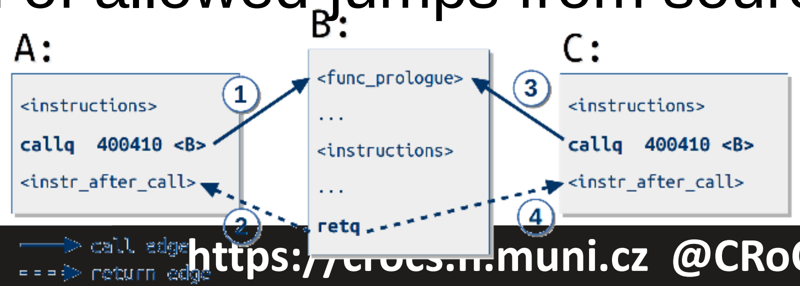
- Follow best practices, standards and coding standards
  - E.g., CERT C Coding Standard https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard
  - (there are many of them, pick for your domain and/or already used in project)

Utilize hardening by compiler and platform

- Attack: Write attacker’s code on stack (e.g., via buffer overflow) and execute it
- Protection: Data Execution Prevention (DEP) – memory pages with non-executable bit set (checked by CPU when using IP)

- Attack: Learn where sensitive info is placed, read from that address (or write)
- Protection: Address Space Layout Randomization (ASLR) – addresses are changed for every program run (hard to predict exact position)

- Attack: Change return address and jump into unexpected functions (Return-oriented programming (ROP))
- Protection: Control flow integrity – build graph of allowed jumps from source code, enforce during runtime



AUTOMATION AND TOOLING

Static vs. dynamic analysis

- Static analysis
  - Static Application Security Testing (SAST)
  - Examine program’s code without executing it
  - Can examine both source code and compiled code
    - source code is easier to understand (more metadata)
  - Can be applied on unfinished code
  - Manual code audit is kind of “static” analysis
- Dynamic analysis
  - Code is executed = program is “running”
  - Input values are supplied, internal memory is examined…
  - Code must compile/run, code coverage by inputs is crucial
- Important: no single tool will ever catch all issues

Automated analysis tools limitations

- Don't expect tools to catch all issues!
- Overall program architecture is not understood
  - sensitivity of program path
  - impact of errors on other parts
- Application semantics is not understood
  - Is string returned to the user? Can string also contain passwords?
- Social context is not understood
  - Who is using the system? High entropy keys encrypted under short guessable password?

Always design for testability

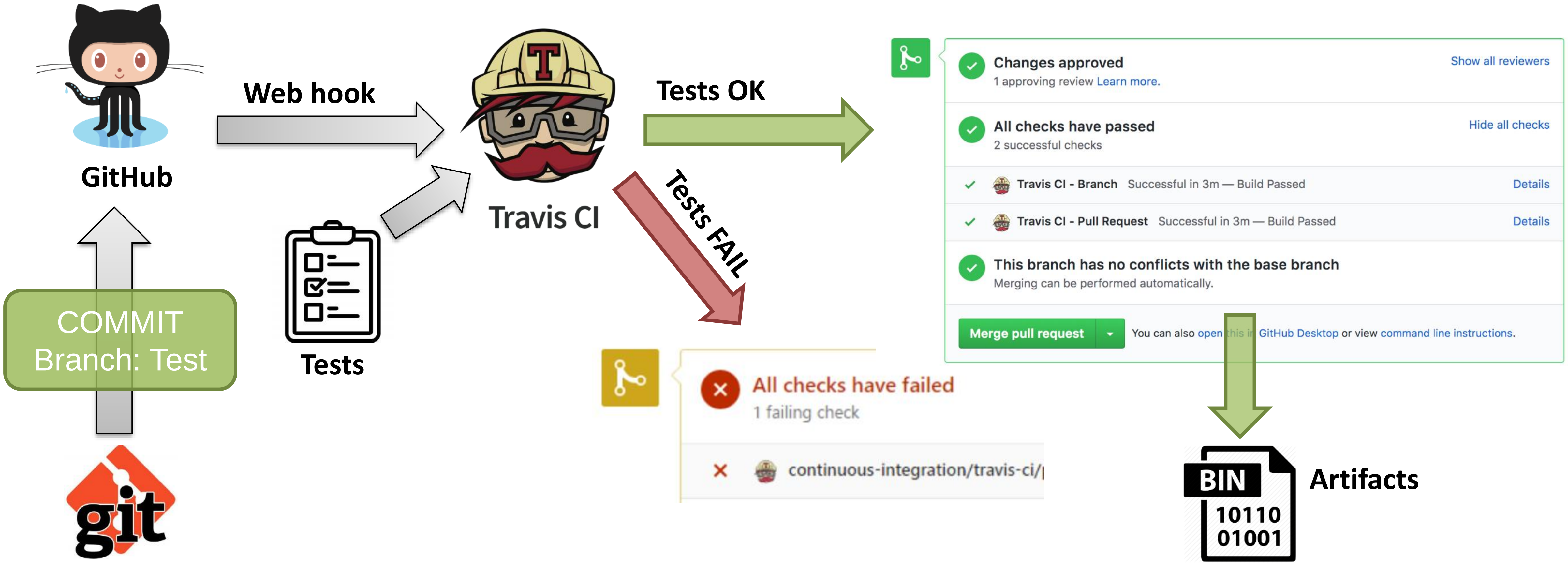
- *“Code that isn't tested doesn't work - this seems to be the safe assumption.”* Kent Beck
- Code written in a way which is easier to test
  - Proper decomposition, unit tests, mock objects
  - Source code annotations (with subsequent analysis)
- Code with extensive quality tests is easier to analyze by static and dynamic tools
- References
  - https://en.wikipedia.org/wiki/Design_For_Test
  - http://www.agiledata.org/essays/tdd.html

CONTINUOUS INTEGRATION

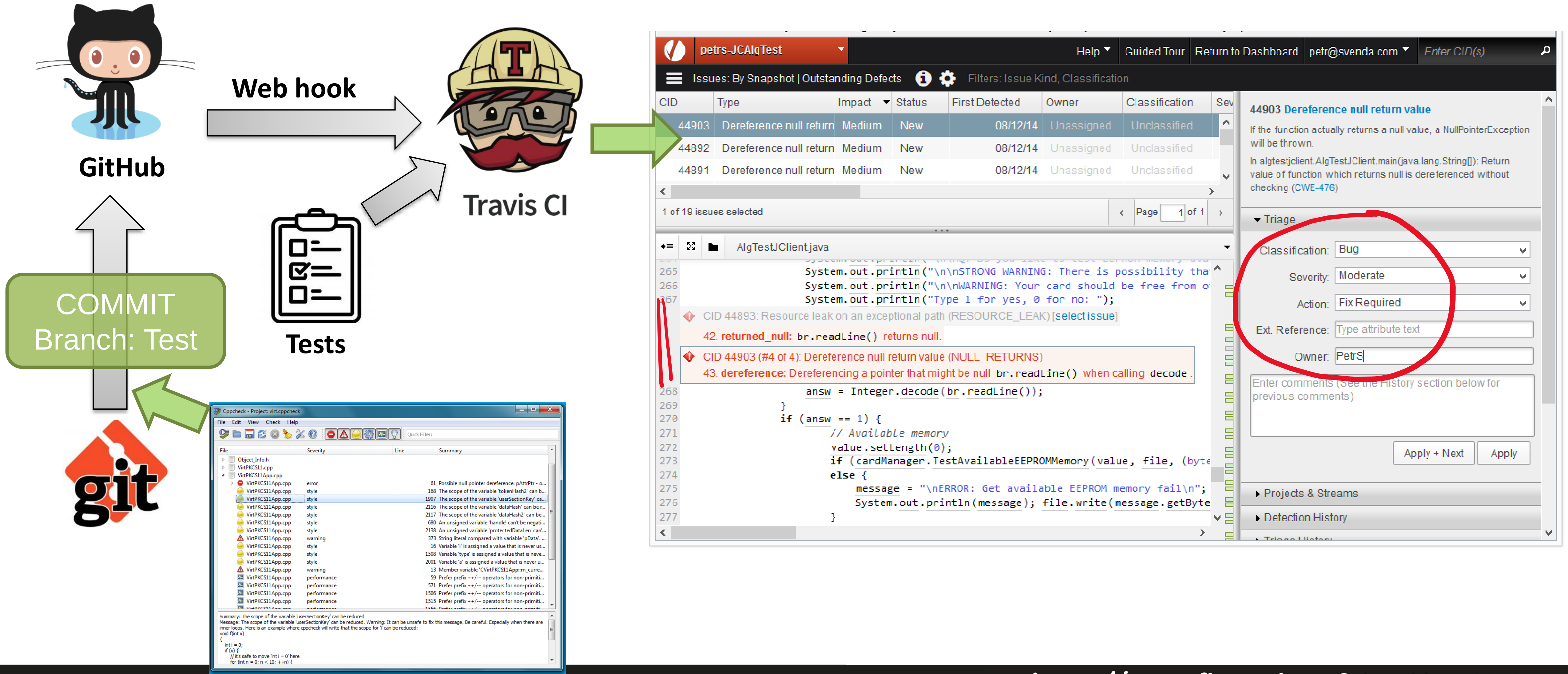
Tests, Continuous integration…

- Running tools manually is insufficient for continuously developed projects
- Include static and dynamic analysis into Continuous Integration process
- Static analysis can be run on unfinished code chunks even before commit
  - On developer side, on commits before merge…
- Dynamic analysis requires sufficient code coverage => quality tests
- Time-consuming analysis can be run "overnight" on server (after push)
  - Or continuously like non-stop fuzzing of the current version of application
- Tools for automatic monitoring of vulnerable components
  - Well-known packages, libraries used by your project with known vulnerability
  - E.g., GitHub's Dependabot

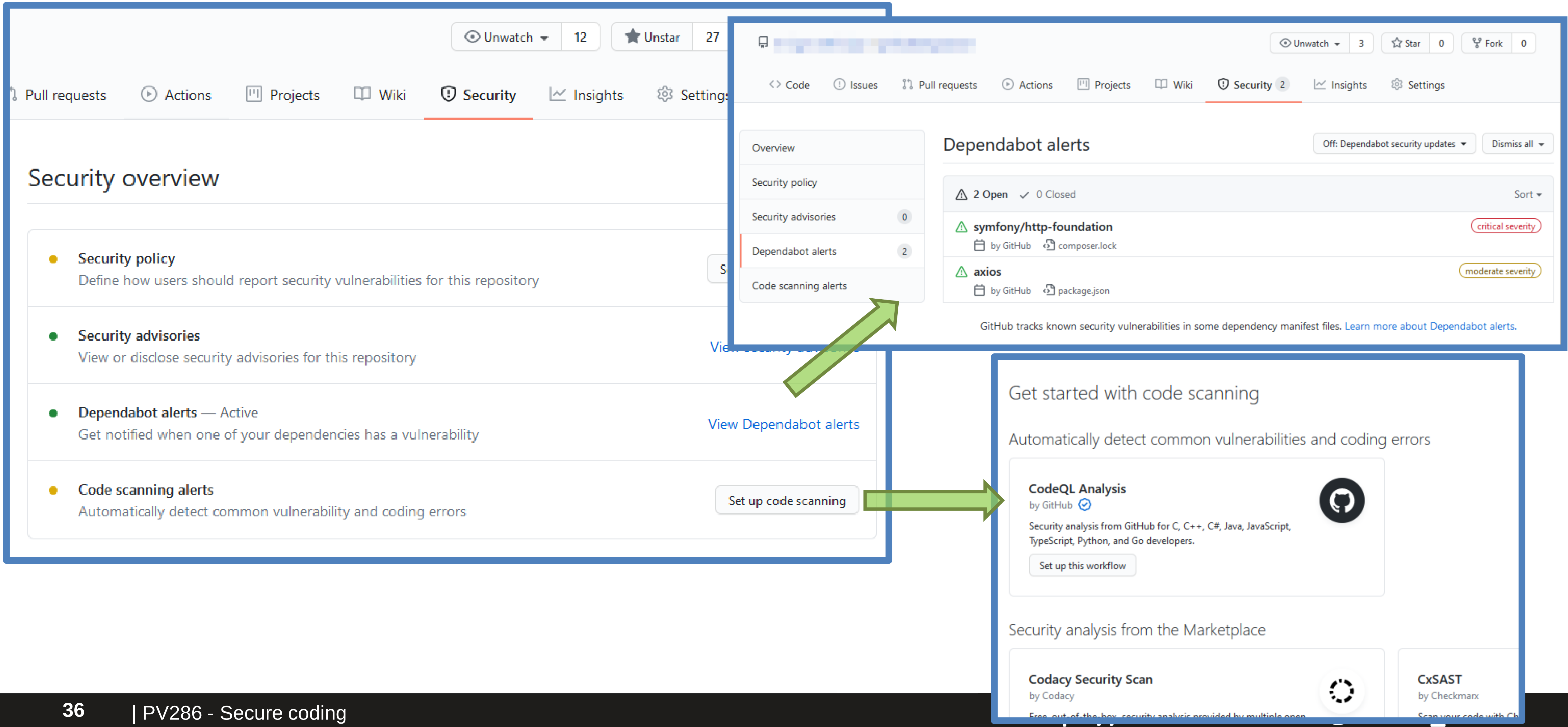
Continuous Integration: GitHub&Travis CI example



CI: adding code analysis (e.g., CppCheck, Coverity)



Dependabot (GitHub)



TYPICAL PROBLEMS FROM REAL WORLD

Typical issues – where theory meets practice ☺

- Insufficient knowledge/education of developers (mature developer would not do majority of issues)
  - Education is time-consuming and expensive (complement with tooling, security champions)
- Legacy code
  - Too many issues reported by tools to fix
  - Fix itself can break things (so developers reluctant to fix what is “not” broken)
- Missing specification of the expected behavior
  - Missing analysis, changing implementation target
  - If implemented code is successful, then is used elsewhere in different condition (original assumptions will be invalidated)
- Adding security only later (“Functionality first!”)
  - It's happening all the time
- Heavy dependance on 3rd party libs
  - No direct control over code, vulnerabilities outside our codebase, possibly unmaintained code (fix means fork)
  - But re-implementing a wheel is usually a worse issue
- Using open-source code can be tricky, you usually must care about:
  - Licenses (tools to help with like Whitesource, Blackduck)
  - Open vulnerabilities, time-to-fix, how active is community
  - In mature organizations, there's usually a open-source governance program that helps developers with choosing the right OSS tools

Typical issues – where theory meets practice ☺

- Human issues
  - No problem before we started to look for them
  - Hard to admit own failures (If I cannot break it, nobody can. “But it is not exploitable”).
  - Unresponsive/threatening companies
  - Same with knowledge, lack of maturity, code guidelines, frameworks
- Security economics
  - Problem is known, yet not fixed – these who need to pay for fix are not these who will suffer
  - Frequently, developer’s KPI is functionality, not security
- Customers do not want to update (new version can break things)
  - Big upgrades mean big risks, small releases/upgrades can help with that
- Trust, but Verify
  - Many companies do not deliver what they promised
  - Security is very common area: insecure updates, insecure installation procedures (curl & chmod & sudo)
- Improper adoption of new tech
  - protobuf, JSON, JWT, serialization...
  - New languages (like "go") are cool, but you need to learn new tooling, test frameworks, CI/CD pipelines, dependencies, ...

- The other side – open-source great tools become also commercial (and free version get semi-abandoned)

Questions

DIGGING DEEPER…

Motivation problem

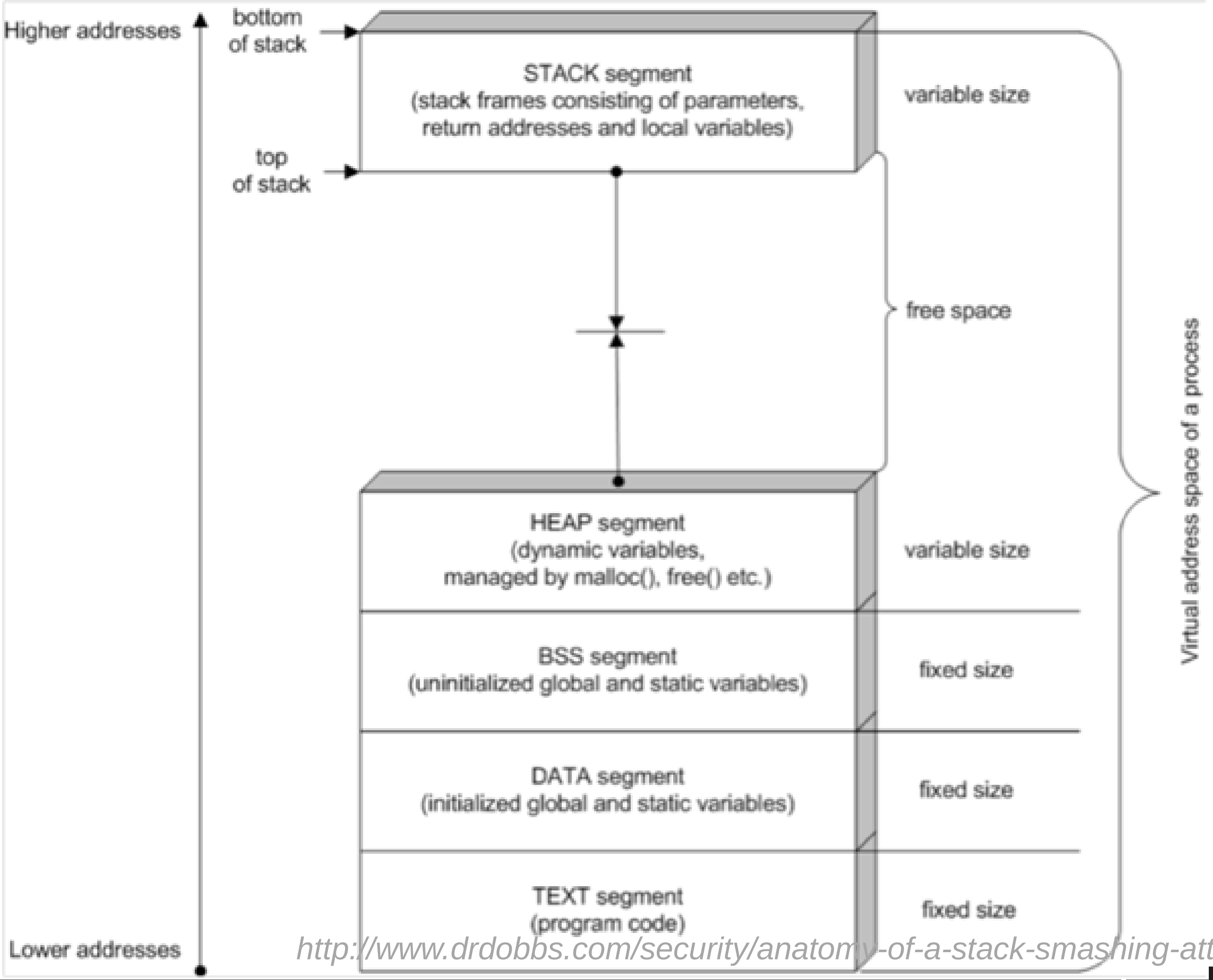
!

- Quiz – what is insecure in given program?
- Can you come up with attack?

```
#define USER_INPUT_MAX_LENGTH  20
char buffer[USER_INPUT_MAX_LENGTH];
bool  isAdmin = false;
gets(buffer);
```

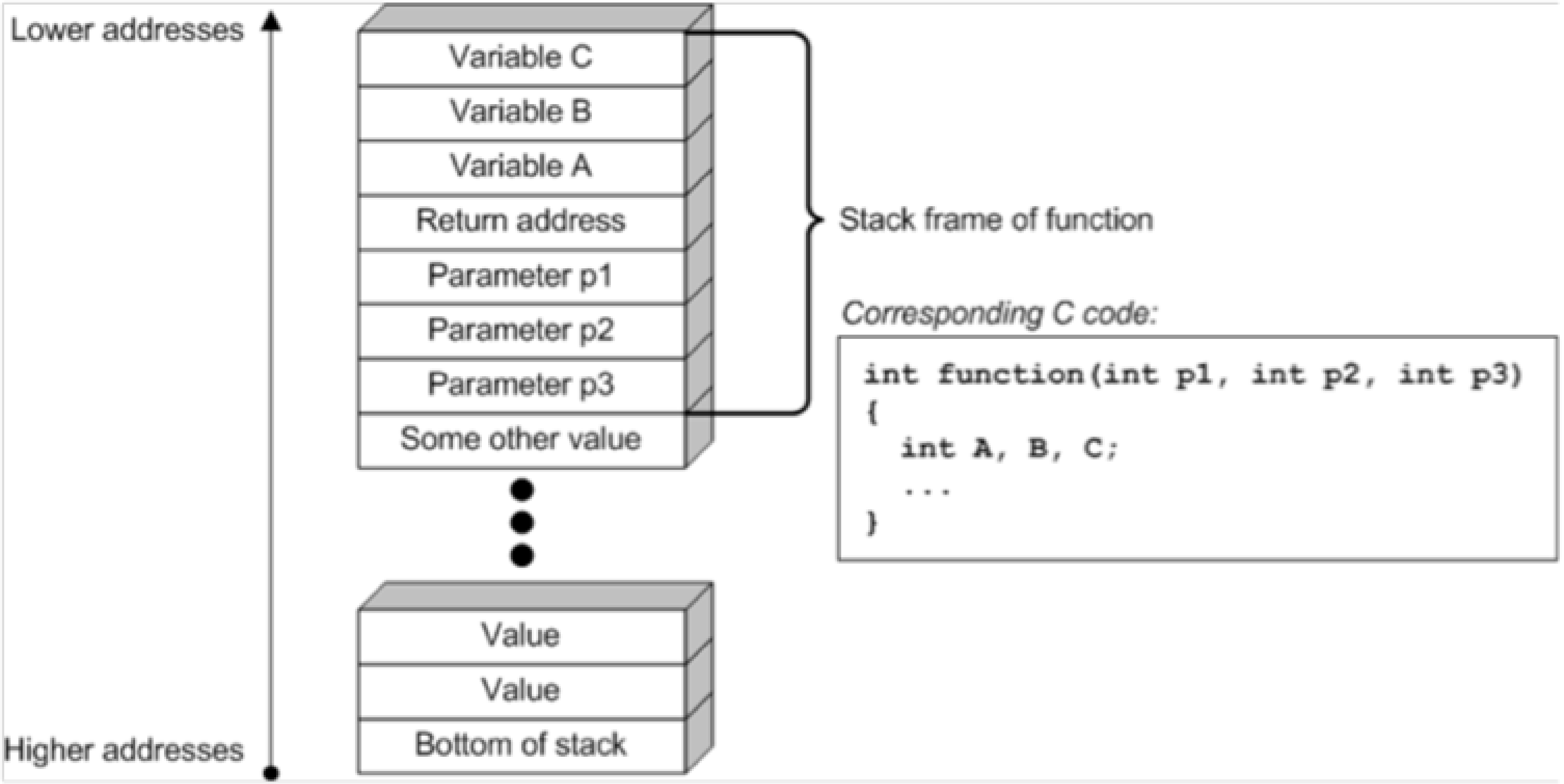
- Classic buffer overflow
- Detailed exploitation demo during labs this week

Process memory layout



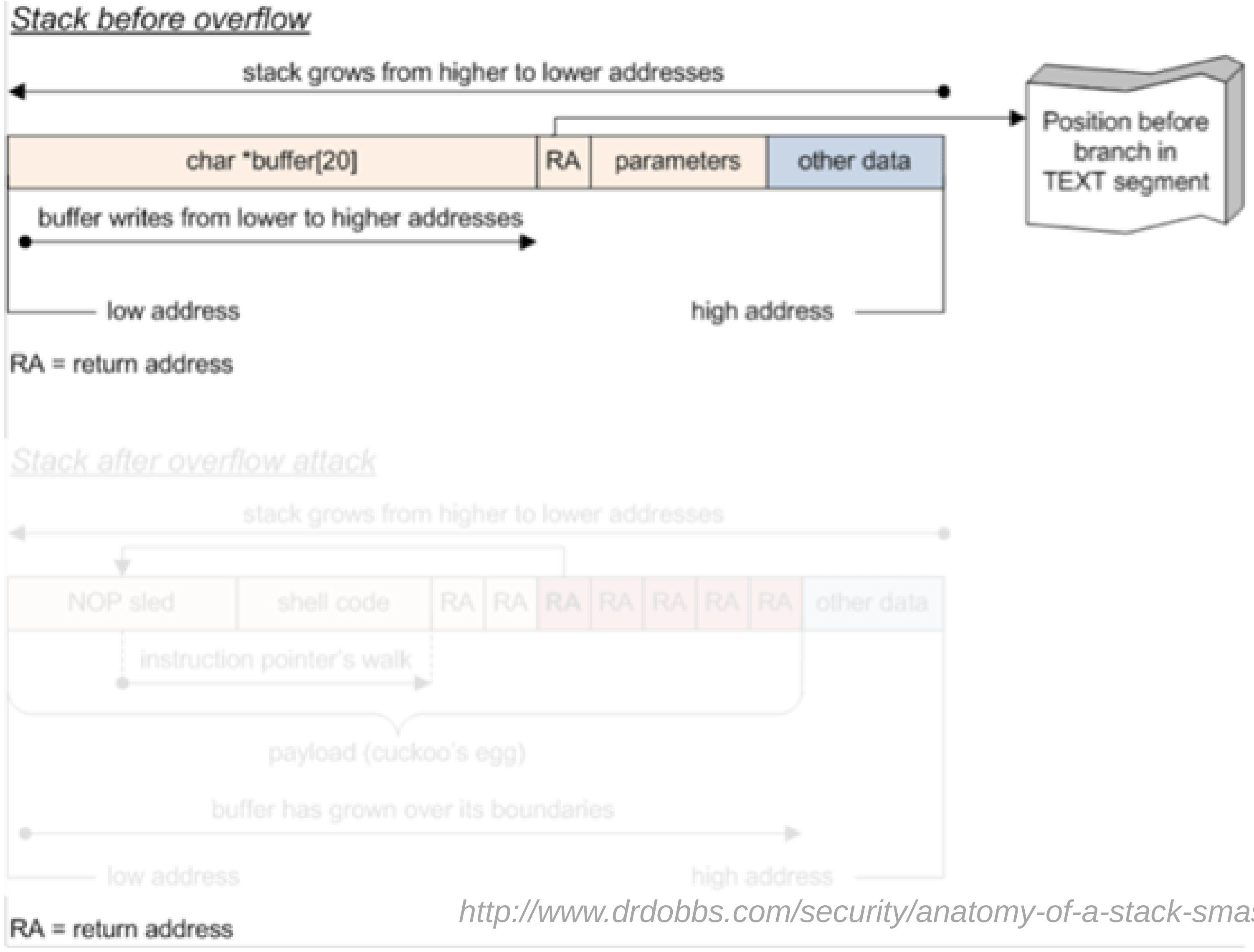
http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#

Stack memory layout



http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#

Stack overflow



http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#

Type-overflow vulnerabilities - motivation

!

- Quiz – what is insecure in given program?
- Can you come up with attack?

```
    for (unsigned char i = 10; i >= 0; i--) {
        /* ... */
    }
```

- And what about following variant?
  - Be aware: char can be both signed (x64) or unsigned (ARM)

```
    for (char i = 10; i >= 0; i--) {
        /* ... */
    }
```

Type overflow – basic problem

- Types are having limited range for the values
  - char: 256 values, int: 2^{32} values
  - add, multiplication can reach lower/upper limit
  - `char value = 250 + 10` == ?
- Signed vs. unsigned types
  - `for (unsigned char i = 10; i >= 0; i--) {/* ... */ }`
- Type value will underflow/overflow
  - CPU overflow flag is set
  - but without active checking not detected in program
- Occurs also in higher-level languages (Java…)

EXAMPLE: MAKE HUGE MONEY WITH TYPE OVERFLOW

Make HUGE money with type overflow

- Bitcoin block 74638 (15th August

```
CBlock(hash=0000000000790ab3, ver=1, hashPrevBlock=0000000000606865, hashMerk
nTime=1281891957, nBits=1c0080
  CTransaction(hash=012cd8, ver=
    CTxIn(COutPoint(000000, -1), c
    CTxOut(nValue= 50.51000000, scriptPubKey=0x4F4BA55D1580F8C3A8A2C7)
  CTransaction(hash=1d5e51, ver=1, vin.size=1, vout.size=2, nLockTime=0)
    CTxIn(COutPoint(237fe8, 0), scriptSig=0xA87C02384E1F184B79C6AC)
    CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HASH160 0xB
    CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HASH160 0x1
  vMerkleTree: 012cd8 1d5e51 618eba

Block hash: 000000                b1470a7ec1c
Transaction hash: 1d5e512a97      2fdac194b65
```

Mining block reward
(was 50BTC at 2010, now smaller)

Input transaction (with 0.5BTC)
https://blockexplorer.com/tx/237fe8348fc77ace11049931058abb034c99698c7fe99b1cc022b1365a705d39

2 output transactions (each with $9*10^{10}$ BTC) !!!

Should have been rejected by miners as value(output) >> value(input), but was not!

Bug dissection

- Bitcoin code uses integer encoding of numbers with fixed position of decimal point (INT64)
  - Smallest fraction of BTC is one Satoshi (sat) = $1/10^8$ BTC
  - 33.54 BTC == 33.54 * 10^8 => 3354000000
- BTW: Why using float numbers is not a good idea?
- CTxOut value:92233720368.54275808 BTC

  = 0x7ffffffffff85ee0
- INT64_MAX = 0x7fffffffffffffff
- Sum of 2 CTx = 0xfffffffffff0bdc0 (overflow)

  = -1000000_{10} = -0.01BTC
  - Difference between input & output interpreted as miner fee

Try this at home!

Type overflow – Bitcoin

```
#include <iostream>
#include <iomanip>
using namespace std;
// Works for Visual Studio compiler, replace __int64 with int64 for other compilers
int main() {
	const __int64 valueMaxInt64 = 0x7fffffffffffffffLL;
	const float COIN = 100000000; // should be __int64 as well, made float for simple printing
	__int64 valueIn = 50000000; // value of input transaction CTxIn
	cout << "CTxIn = " << valueIn / COIN << endl;
	__int64 valueOut1 = 9223372036854275808L; // first out
	cout << "CTxOut1 = " << valueOut1 / COIN << endl;
	__int64 valueOut2 = 9223372036854275808L; // second out
	cout << "CTxOut2 = " << valueOut2 / COIN << endl;

	__int64 valueOutSum = valueOut1 + valueOut2;   // sum which overflow
	cout << "CTxOut sum = " << valueOutSum / COIN << endl;
	// Difference between input and output is interpreted as fee for a miner (0.01 BTC)
	__int64 fee = valueIn - valueOutSum;
	cout << "Miner fee = " << fee / COIN << endl;
	return 0;
}
```

BugFix – proper checking for overflow

https://github.com/bitcoin/bitcoin/commit/d4c6b90ca3f9b47adb1b2724a0c3514f80635c84#diff-118fcbaaba162ba17933c7893247df3aR1013

11 main.h

```diff
@@ -18,6 +18,7 @@ static const unsigned int MAX_SIZE = 0x02000000;
 static const unsigned int MAX_BLOCK_SIZE = 1000000;
 static const int64 COIN = 100000000;
 static const int64 CENT = 1000000;
+static const int64 MAX_MONEY = 21000000 * COIN;
 static const int COINBASE_MATURITY = 100;

 static const CBigNum bnProofOfWorkLimit(~uint256(0) >> 32);
@@ -471,10 +472,18 @@ class CTransaction
         if (vin.empty() || vout.empty())
             return error("CTransaction::CheckTransaction() : vin or vout empty");

-        // Check for negative values
+        // Check for negative or overflow output values
+        int64 nValueOut = 0;
         foreach(const CTxOut& txout, vout)
+        {
             if (txout.nValue < 0)
                 return error("CTransaction::CheckTransaction() : txout.nValue negative");
+            if (txout.nValue > MAX_MONEY)
+                return error("CTransaction::CheckTransaction() : txout.nValue too high");
+            nValueOut += txout.nValue;
+            if (nValueOut > MAX_MONEY)
+                return error("CTransaction::CheckTransaction() : txout total too high");
+        }

         if (IsCoinBase())
         {
```

Try this at home!

Questions

- When exactly overflow happens?
- Why mining reward was 50.51 and not exactly 50?
  - CTxOut(nValue= 50.51000000
- How to check for type overflow?

SOURCE CODE PROTECTIONS

COMPILER PROTECTIONS

PLATFORM PROTECTIONS

Safe add and mult operations in C/C++

- Compiler-specific non-standard extensions of C/C++
- GCC: __builtin_add_overflow, __builtin_mul_overflow …

```
bool __builtin_add_overflow (type1 a, type2 b, type3 *res)
```

  - Result returned as third (pointer passed) argument
  - Returns true if overflow occurs
  - https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html
- MSVC: SafeInt wrapper template (for int, char…)
  - Overloaded all common operations (drop in replacement)
  - Returns SafeIntException if overflow/underflow
  - https://learn.microsoft.com/en-us/cpp/safeint/safeint-library?view=msvc-170

```
#include <safeint.h>
using namespace msl::utilities;
SafeInt<int> c1 = 1; SafeInt<int> c2 = 2;
```

```
// Normal use
c1 = c1 + c2;
```

Safe add and mult operations in Java

- Java SE 8 introduces extensions to java.lang.Math
- ArithmeticException thrown if overflow/underflow

```
public static int addExact(int x, int y)
public static long addExact(long x, long y)
public static int decrementExact(int a)
public static long decrementExact(long a)
public static int incrementExact(int a)
public static long incrementExact(long a)
public static int multiplyExact(int x, int y)
public static long multiplyExact(long x, long y)
public static int negateExact(int a)
public static long negateExact(long a)
public static int subtractExact(int x, int y)
public static long subtractExact(long x, long y)
public static int toIntExact(long value)
```

Format string vulnerabilities - motivation

- Quiz – what is insecure in given program?
- Can you come up with attack?

```
int main(int argc, char * argv[]) {
    printf(argv[1]);
    return 0;
}
```

Format string vulnerabilities

- Wide class of functions accepting format string
  - `printf("%s", X);`
  - resulting string is returned to user (= potential attacker)
  - formatting string can be under attacker's control
  - variables formatted into string can be controlled
- Resulting vulnerability
  - memory content from stack is formatted into string
  - possibly any memory if attacker control buffer pointer

Information disclosure vulnerabilities

- Exploitable memory vulnerability leading to read access (not write access)
  - attacker learns some information from the memory
- Direct exploitation
  - secret information (cryptographic key, password...)
- Precursor for next step (very important with DEP&ASLR)
  - module version
  - current memory layout after ASLR (stack/heap pointers)
  - stack protection cookies (/GS)

Format string vulnerability - example

!

- Example retrieval of security cookie and return address

```
int main(int argc, char* argv[]) {
  char buf[64] = {};
  sprintf(buf, argv[1]);
  printf("%s\n", buf);
  return 0;
}
```

argv[1] submitted by an attacker
E.g., %x%x%x….%x
Stack content is printed
Including security cookie and RA

Don’t let user/attacker to provide own formatting strings

Non-terminating functions - example

!

- What is wrong with following code?

```
int main(int argc, char* argv[]) {
    char buf[16];
    strncpy(buf, argv[1], sizeof(buf));
    return printf("%s\n",buf);
}
```

strncpy - manual

function

strncpy

<cstring>

```
char * strncpy ( char * destination, const char * source, size_t num );
```

Copy characters from string

Copies the first *num* characters of *source* to *destination*. If the end of the *source* C string (which is signaled by a null-character) is found before *num* characters have been copied, *destination* is padded with zeros until a total of *num* characters have been written to it.

No null-character is implicitly appended at the end of *destination* if *source* is longer than *num*. Thus, in this case, *destination* shall not be considered a null terminated C string (reading it as such would overflow).

destination and *source* shall not overlap (see memmove for a safer alternative when overlapping).

Parameters

`destination`
: Pointer to the destination array where the content is to be copied.

`source`
: C string to be copied.

`num`
: Maximum number of characters to be copied from *source*.
size_t is an unsigned integral type.

http://www.cplusplus.com/reference/cstring/strncpy/?kw=strncpy

Non-terminating functions for strings

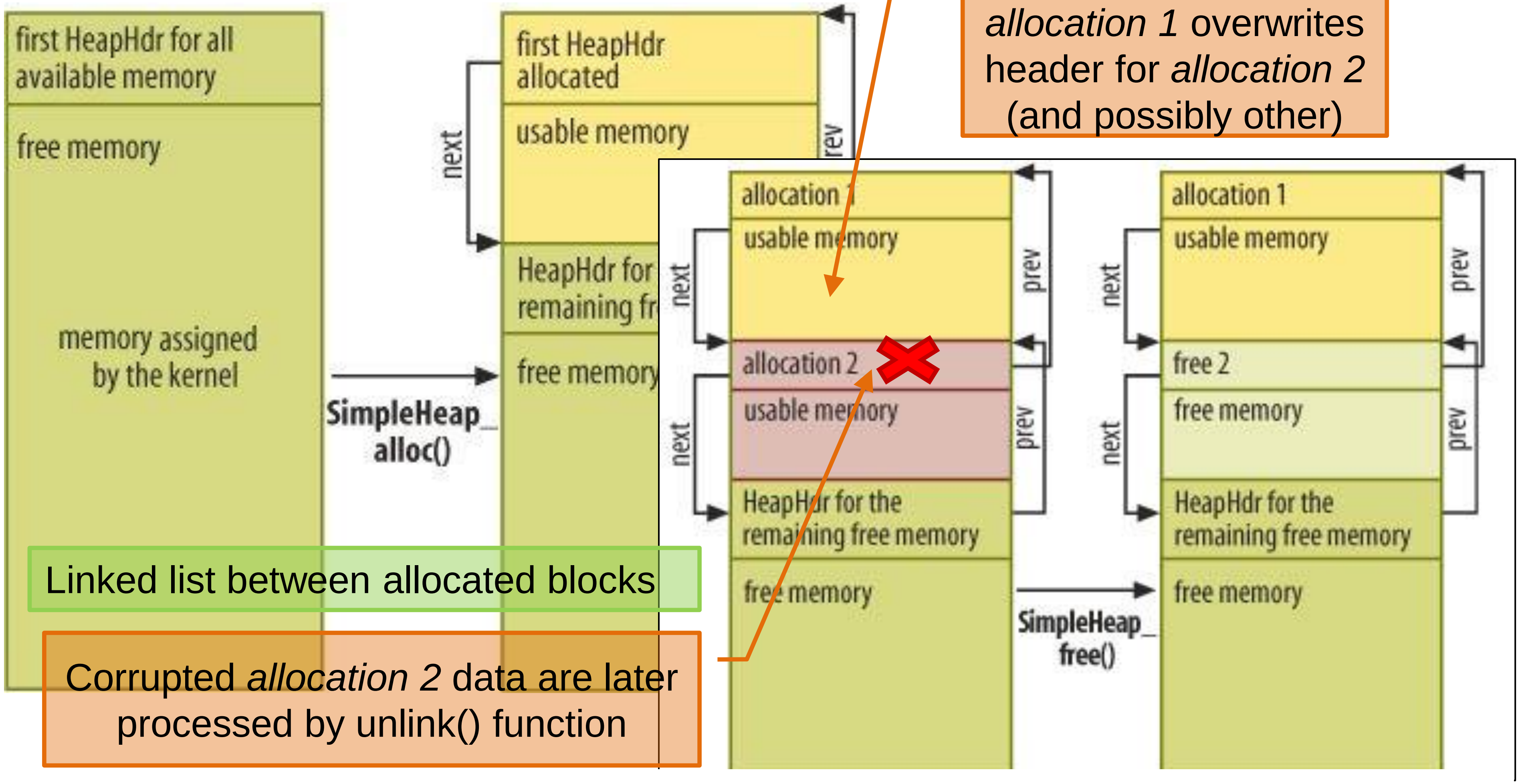
- strncpy
- snprintf
- vsnprintf
- mbstowcs
- MultiByteToWideChar

- wcsncpy
- snwprintf
- vsnwprintf
- wcstombs
- WideCharToMultiByte

- Non-null terminated Unicode string more dangerous
  - C-string processing stops on first zero
  - any binary zero (ASCII)
  - 16-bit aligned wide zero character (UNICODE)

Null termination specific for C, but terminating/separating characters relevant in any other language

Heap overflow



Buffer overflow in *allocation 1* overwrites header for *allocation 2* (and possibly other)

Linked list between allocated blocks

Corrupted *allocation 2* data are later processed by unlink() function

Felix "FX" Lindner, http://www.h-online.com/security/features/A-Heap-of-Risk-747220.html

Secure C library – selected functions

```
char *gets(
   char *buffer
);

char *gets_s(
   char *buffer,
   size_t sizeInCharacters
);
```

- Formatted input/output functions
  - **gets_s**
  - **scanf_s**, wscanf_s, **fscanf_s**, fwscanf_s, sscanf_s, swscanf_s, vfscanf_s, vfwscanf_s, vscanf_s, vwscanf_s, vsscanf_s, vswscanf_s
  - **fprintf_s**, fwprintf_s, **printf_s**, printf_s, snprintf_s, snwprintf_s, **sprintf_s**, swprintf_s, vfprintf_s, vfwprintf_s, vprintf_s, vwprintf_s, vsnprintf_s, vsnwprintf_s, vsprintf_s, vswprintf_s
  - functions take additional argument with buffer length
- File-related functions
  - tmpfile_s, tmpnam_s, fopen_s, freopen_s
    - takes pointer to resulting file handle as parameter
    - return error code

Secure C library – selected functions

- Environment, utilities
  - getenv_s, wgetenv_s
  - bsearch_s, qsort_s
- Memory copy functions
  - memcpy_s, memmove_s, strcpy_s, wcscpy_s, strncpy_s, wcsncpy_s
- Concatenation functions
  - strcat_s, wcscat_s, strncat_s, wcsncat_s
- Search functions
  - strtok_s, wcstok_s
- Time manipulation functions...

Secure C library

- Secure versions of commonly misused functions
  - bounds checking for string handling functions
  - better error handling
- Also added to new C standard ISO/IEC 9899:2011
- Microsoft Security-Enhanced Versions of CRT Functions
  - MSVC compiler issue warning C4996, more functions then in C11
- Secure C Library
  - http://docwiki.embarcadero.com/RADStudio/XE3/en/Secure_C_Library
  - https://docs.microsoft.com/en-us/cpp/c-runtime-library/security-enhanced-versions-of-crt-functions
  - https://docs.microsoft.com/en-us/cpp/c-runtime-library/security-features-in-the-crt
  - http://www.drdobbs.com/cpp/the-new-c-standard-explored/232901670

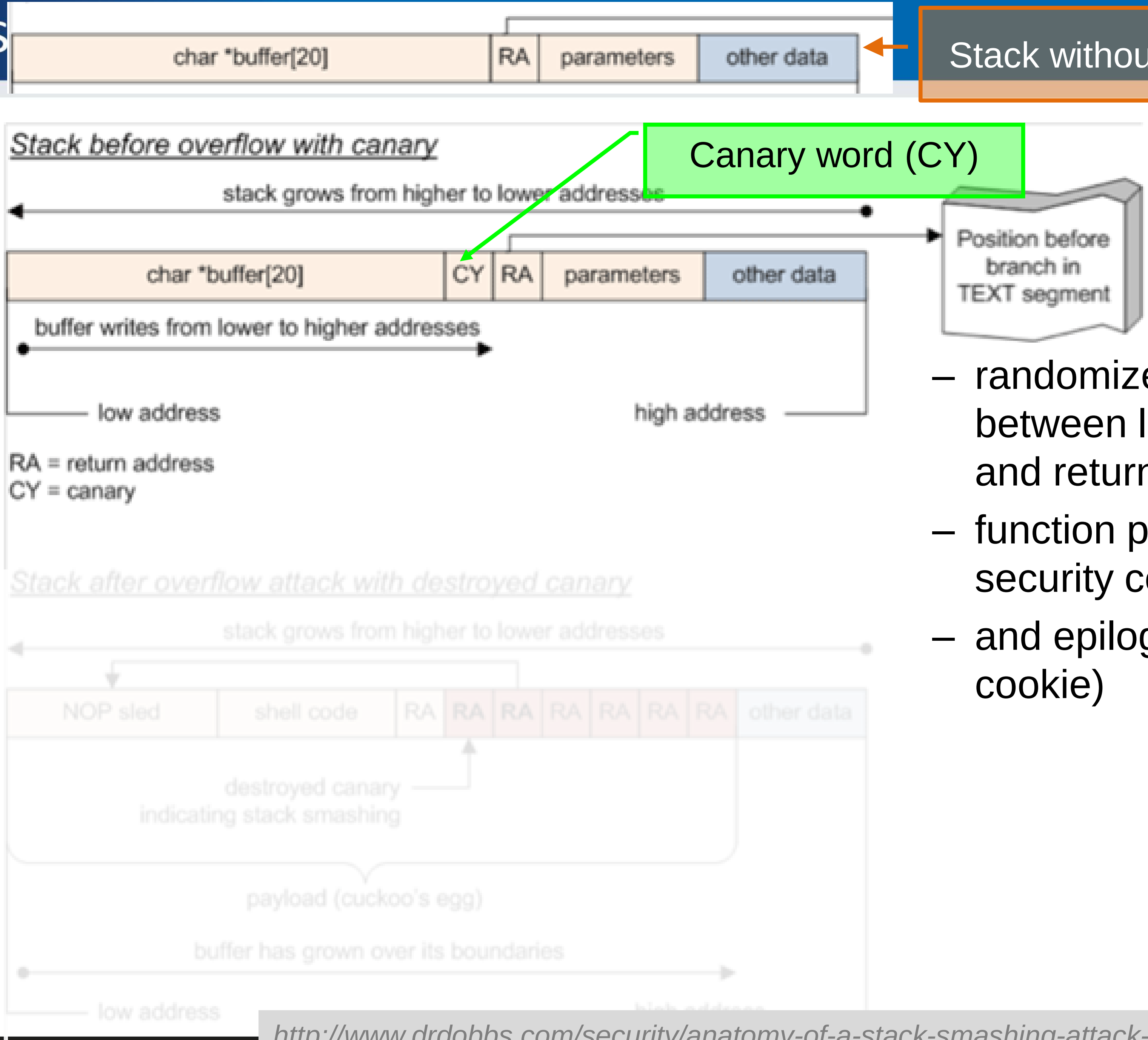
SOURCE CODE PROTECTIONS

COMPILER PROTECTIONS

PLATFORM PROTECTIONS

Stack without canary word

Canary word (CY)



!

- randomized cookie between local variables and return address
- function prolog (add security cookie)
- and epilog (check cookie)

http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#

MSVC Compiler security flags - /GS

- /GS switch (added from 2003, improves in time)
  - http://msdn.microsoft.com/en-us/library/8dbf701c.aspx
  - multiple different protections against buffer overflow
  - mostly focused on stack protection
- /GS protects:
  - return address of function
  - address of exception handler
  - vulnerable function parameters (arguments)
  - some of the local buffers (GS buffers)

/GS is applied in both DEBUG and RELEASE modes

- /GS protection is (automatically) added only when needed
  - to limit performance impact, decided by compiler (/GS rules)
  - `#pragma strict_gs_check(on)` - enforce strict rules application

/GS – what is NOT protected

- /GS compiler option does not protect against all buffer overrun security attacks
- Corruption of address in vtable
  - (table of addresses for virtual methods)
- Example: buffer and a vtable in an object, a buffer overrun could corrupt the vtable
- Functions with variable arguments list (...)

Automatic tools add vital protections, but are NOT replacement for secure defensive programming

GCC compiler - StackGuard & ProPolice

!

- StackGuard released in 1997 as extension to GCC
  - but never included as official buffer overflow protection
- GCC Stack-Smashing Protector (ProPolice)
  - patch to GCC 3.x
  - included in GCC 4.1 release
  - **`-fstack-protector`** (string protection only)
  - **`-fstack-protector-all`** (protection of all types)
  - on some systems enabled by default (OpenBSD)
    - **`-fno-stack-protector`** (disable protection)

Example: Stack canary

GCC -fstack-protector-all

```
vuln:
.LFB0:
    .cfi_startproc
    pushq   %rbp                ; current base pointer onto stack
    .cfi_def_cfa_offset 16
    movq    %rsp, %rbp          ; stack pointer becomes new base pointer
    .cfi_offset 6, -16
    .cfi_def_cfa_register 6
    subq    $48, %rsp           ; reserve space for
                                    ; local variables on stack

        ; bring arguments from registers onto stack
    movq    %rdi, -40(%rbp)     ; 1st argument from rdi to stack

        ; SSP's prolog: put canary onto stack
    movq    %fs:40, %rax        ; canary from %fs:40 to rax
    movq    %rax, -8(%rbp)      ; canary from rax onto stack
    xorl    %eax, %eax          ; set rax to zero

        ; prepare parameters for strcpy()
    movq    -40(%rbp), %rdx     ; 1st argument to rdx
    leaq    -32(%rbp), %rax     ; 2nd argument to rax

        ; call strcpy()
    movq    %rdx, %rsi          ; source address from rdx to rsi
    movq    %rax, %rdi          ; destination address from rax to rdi
    call    strcpy              ; call strcpy()

        ; SSP's epilog: check canary
    movq    -8(%rbp), %rax      ; canary from stack to rax
    xorq    %fs:40, %rax        ; original canary XOR rax
    je   .L3                  ; if no overflow -> XOR results in zero
                                    ;                  => jump to label .L3
                                    ; if overflow     -> XOR results in non-zero
    call    __stack_chk_fail    ;                => call __stack_chk_fail()

.L3:
    leave                       ; clean-up stack
    ret                         ; return
    .cfi_endproc
```

```c
#include <string.h>

void vuln(const char *str)
{
  char buf[20];
  strcpy(buf, str);
}

int main(int argc, char *argv[])
{
  vuln(argv[1]);
  return 0;
}
```

ATTACKER

Can an attacker still bypass stack canary?

How to bypass stack protection cookie?

!

- Scenario:
  - long-term running of daemon on server
  - no exchange of cookie between calls

1. Obtain security cookie by one call
   - cookie is now known and can be incorporated into stack-smashing data
2. Use second call to change only the return address

What attacker can do with changed return address?

SOURCE CODE PROTECTIONS

COMPILER PROTECTIONS

PLATFORM PROTECTIONS

Data Execution Prevention (DEP)

!

- *Motto: When boundary between code and data blurs (buffer overflow, SQL injection…) then exploitation might be possible*
- Data Execution Prevention (DEP)
  - prevents application to execute code from non-executable memory region
  - available in modern operating systems
    - Linux > 2.6.8, WinXPSP2, Mac OSX, iOS, Android…
  - difference between ‘hardware’ and ‘software’ based DEP

Hardware DEP

- Supported from AMD64 and Intel Pentium 4
  - OS must add support of this feature (around 2004)
- CPU marks memory page as non-executable
  - most significant bit (63th) in page table entry (NX bit)
  - 0 == execute, 1 == data-only (non-executable)
- Protection typically against buffer overflows
- Cannot protect against all attacks!
  - e.g., code compiled at runtime (produced by JIT compiler) must have both instructions and data in executable page
  - attacker redirect execution to generated code (JIT spray)
  - used to bypass Adobe PDF and Flash security features

Software “DEP”

- Unrelated to NX bit (no CPU support required)
- When exception is raised, OS checks if exception handling routine pointer is in executable area
  - – Microsoft’s Safe Structured Exception Handling
- Software DEP is not preventing general execution in non-executable pages
  - – different form of protection than hardware DEP

Address Space Layout Randomization (ASLR)

!

- Random reposition of executable base, stack, heap and libraries address in process's address space
  - aim is to prevent exploit to reliably jump to required address
- Performed every time a process is loaded into memory
  - random offset added to otherwise fixed address
  - applies to program and also dynamic libraries
  - entropy of random offset is important (bruteforce)
- Operating System kernel ASLR (kASLR)
  - more problematic as long-running (random, but fixed until reboot)
- Introduced by Memco software (1997)
  - fully implemented in Linux PaX patch (2001)
  - MS Vista, enabled by default (2007), MS Win 8 more entropy (2012)

ASLR – impact on attacks

- ASLR introduced big shift in attacker mentality
- Attacks are now based on gaps in ASLR
  - legacy programs/libraries/functions without ASLR support
    - ! /DYNAMICBASE
  - address space spraying (heap/JIT)
  - predictable memory regions, insufficient entropy

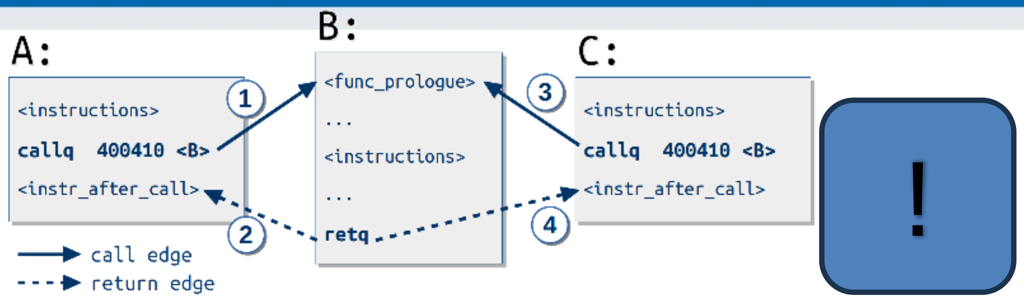
Can attacker execute desired functionality without changing code?

Return-oriented programming (ROP)

!

- Return-into-library technique (Solar Designer, 1997)
  - method for bypassing DEP
  - no write of attacker's code to stack (as is prevented by DEP)
  1. function return address replaced by pointer to standard library function
  2. library function arguments replaced according to attackers needs
  3. function return results in execution of library function and given arguments
  - Example: system call wrappers like system()
- Borrowed code chunks
  - Problem: 64-bit hardware introduced different calling convention
    - first arguments to function passed in CPU registers instead of via stack
  - attacker tries to find instruction sequences from any function that pop values from the stack into registers (automated search by ROPgadget)
  - necessary arguments are inserted into registers
  - return-into-library attack is then executed as before

Control flow integrity



- Promising technique with low overhead
- Classic CFI (2005), Modular CFI (2014)
  - avg 5% impact, 12% in worst case
  - part of LLVM C compiler (CFI usable for other languages as well)

1. Analysis of source code to establish control-flow graph (which function can call what other functions)
2. Assign shared labels between valid caller X and callee Y
3. When returning into function X, shared label is checked
4. Return to other function is not permitted

https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-carlini.pdf

DEP and ASLR should be combined

!

- *“For ASLR to be effective, DEP/NX must be enabled by default too.”* M. Howard, Microsoft
- /GS combined with /DYNAMICBASE and /NXCOMPAT
  - /NXCOMPAT (==DEP)
  - prevents insertion of new attacker's code and forces ROP
  - /DYNAMICBASE (==ASLR) randomizes code chunks utilized by ROP
  - /GS prevents modification of return pointer used later for ROP
  - /DYNAMICBASE randomizes position of master cookie for /GS
- Visual Studio → Configuration properties →
  - Linker → All options
  - C/C++ → All options

SUMMARY

Mandatory reading

- SANS: 2017 State of Application Security
  - https://web.archive.org/web/20180119191652/https://www.sans.org/reading-room/whitepapers/application/2017-state-application-security-balancing-speed-risk-38100
  - Which applications are of main security concern?
  - What is expected time to deploy patch for critical security vulnerability?
  - How does your organization test applications for vulnerabilities?
  - Which language is the most common source of security risk?

Optional reading

- Marcel Böhme: “Guarantees in Software Security”
  - An article from Ferbuary 2024: https://arxiv.org/abs/2402.01944
  - Interesting read with many practical examples. However, it is academic and might not be detailed enough (e.g., if you never heard about a particular bug, then it is hard to follow since it is not explained in detail).
  - “We review general approaches to reason about the security of a software system and reflect upon the guarantees they provide. We introduce a taxonomy of fundamental challenges towards the provision of guarantees, and discuss how these challenges are routinely exploited to attack a system in spite of credible assurances about the absence of such bugs. “

Questions