Software Reliability and Security

Module 10

Winter 2017

Based on ACM SAC 2011 Tutorial
(Shahriar and M. Zulkernine, 2011)

Presentation/Lecture Schedule and Report Due Dates

- Presentation 1
  - Related background paper
  - Jan 27, Feb 1, 3
- Presentation 2
  - Project proposal
  - March 1, 3, 8
- Presentation 3
  - Final project report
  - March 24, 29, 31

- Lectures
  Jan 13, 18, 20, 25, 27
  Feb 1, 3, 8, 10, 15, 17
  March 1, 3, 8, 10, 15, 17, 22, 24, 29, 31

- Project Proposal Due
  Tuesday, February 28

- Final Project Report Due
  Monday, April 10

- Final Exam
  Wednesday, April 12, 10:00am

Background

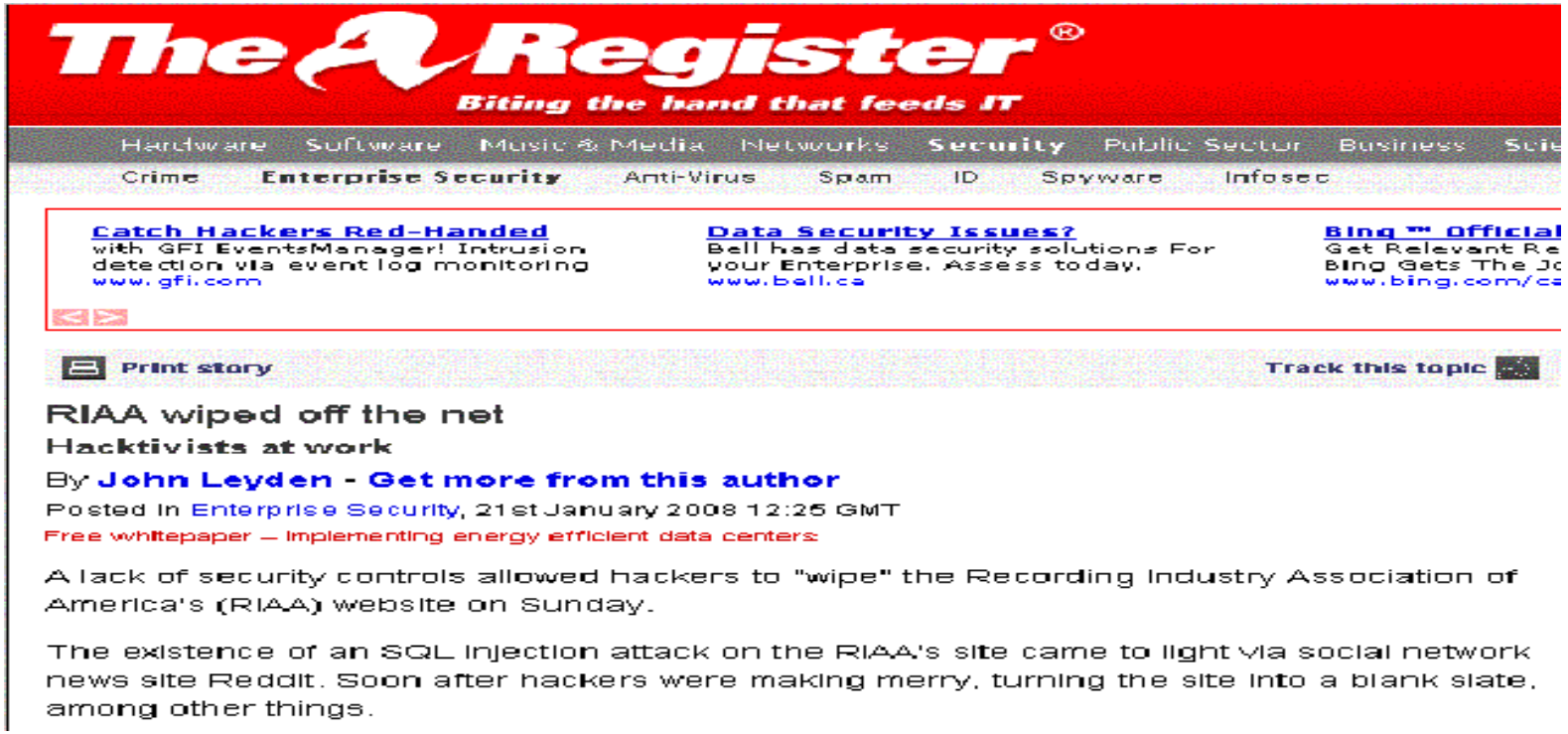
- Programs are accessible to both legitimate users and hackers
- Ideally, a program should conform to expected behaviors
  - Provide desired outputs based on supplied inputs
- In practice, a program's behavior is unexpected
  - Program crashes or leaks sensitive information

Unexpected program behavior

- A very long URL in the address bar of Microsoft Internet Explorer (6.0) (Source: CVE 2006-3869)
  - Cause
    - Internet Explorer fails to due to buffer overflow
  - Impact
    - With a specially crafted request, an attacker can execute arbitrary code

Source: Common Vulnerabilities and Exposures (CVE) http://cve.mitre.org

Information deletion



The Register®
Biting the hand that feeds IT

Hardware Software Music & Media Networks Security Public Sector Business Scie
Crime Enterprise Security Anti-Virus Spam ID Spyware Infosec

Catch Hackers Red-Handed
with GFI EventsManager! Intrusion detection via event log monitoring
www.gfi.com

Data Security Issues?
Bell has data security solutions For your Enterprise. Assess today.
www.bell.ca

Bing ™ Official
Get Relevant Re
Bing Gets The Jo
www.bing.com/cs

Print story Track this topic

RIAA wiped off the net
Hacktivists at work
By John Leyden - Get more from this author
Posted in Enterprise Security, 21st January 2008 12:25 GMT
Free whitepaper — implementing energy efficient data centers

A lack of security controls allowed hackers to "wipe" the Recording Industry Association of America's (RIAA) website on Sunday.

The existence of an SQL injection attack on the RIAA's site came to light via social network news site Reddit. Soon after hackers were making merry, turning the site into a blank slate, among other things.

"The existence of an SQL injection attack on the RIAA's site came to light via social network news site … hackers turned the site into a blank slate, among other things."

"The RIAA has restored RIAA.org, although whether it's any more secure than before remains open to question" - January 2008

Source: http://www.theregister.co.uk/2008/01/21/riaa_hacktivism/

Information modification

XSS flaw makes PM say: "I want to suck your blood"

By Liam Tung, ZDNet Australia | 2007/10/09 16:52:02

Tags: cross-site scripting, flaw, labor, liberal, security, xss

Change the text size: a | A

Print this | E-mail this | Leave a comment | Clip this

The Web sites of Australia's two major political parties contain cross-site scripting (XSS) flaws, which could be exploited to fraudulently acquire political donations, say security experts.

A short line of script developed by a security enthusiast, Bsoric, causes the Liberal Party's Web site to read: "John Howard says: I want to suck your blood", while another script caused a window to pop up on the Labor Party's Web site, urging viewers to "Vote Liberal!"

Carl Jongsma, security expert at Sunnet Beskerming, said although the vulnerabilities on each party's Web sites have been exploited for comedic purposes, it would be possible to use the script to fraudulently target people for political donations.

"The Web sites of Australia's two major political parties contain cross-site scripting (XSS) flaws" - October 2007

Source: http://www.builderau.com.au/news/soa/XSS-flaw-makes-PM-say-I-want-to-suck-your-blood-/0,339028227,339282682,00.htm

More list of incidents

- Common Vulnerabilities and Exposures (CVE)
  http://cve.mitre.org
- Open Source Vulnerability Database (OSVDB),
  http://osvdb.org
- Web-Application Security Consortium (WASC)
  http://www.webappsec.org/projects/whid/
  - Attack method
  - Outcome
  - Location
  - …

What is wrong in these programs?

- These programs are developed and tested by the brightest engineers in the industry
- Some interesting aspects might be missing
  - Program security vulnerability mitigations in pre and post deployment phases
  - Ensure that programs cannot be exploited with malicious inputs

Outline

- Provide an overview of common program security vulnerability exploitations (Part I)
- Introduce mitigation techniques (Part II)
  - Program security testing
  - Static analysis, Monitoring, Hybrid analysis

Program security vulnerability

- Vulnerabilities are flaws in programs that allow attackers to expose or alter sensitive information (Dowd et al. 2007)
- Exploitation of vulnerabilities are attacks
- Common examples of vulnerabilities
  - Buffer Overflow (BOF)
  - Format String Bug (FSB)
  - SQL Injection (SQLI)
  - Cross Site Scripting (XSS)

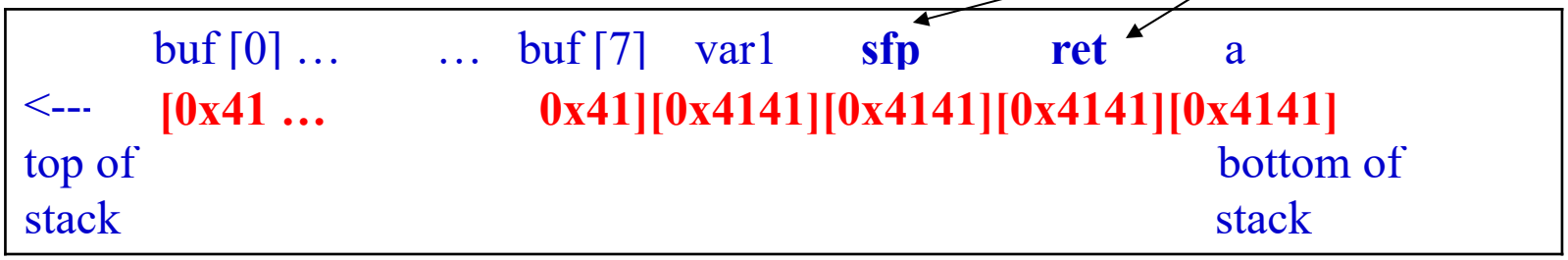
Buffer overflow (BOF)

- Specific flaws in implementations that allow attackers to overflow data buffers
- Consequence of BOF attacks
  - Program state corruption
  - Program crash due to overwriting of sensitive neighboring variables such as return addresses

Buffer overflow example

```
1.  void foo (int a) {
2.        int var1;
3.        char buf [8];
4.        strcpy (buf, "AAAAAAAAAAAAAA");
          …
          return;
    }
```

C code snippet of *foo* function (adapted from [2])



Stack layout of *foo* function (adapted from [2])

sfp: Stack frame pointer. ret: return address of *foo* function

Format string bug (FSB)

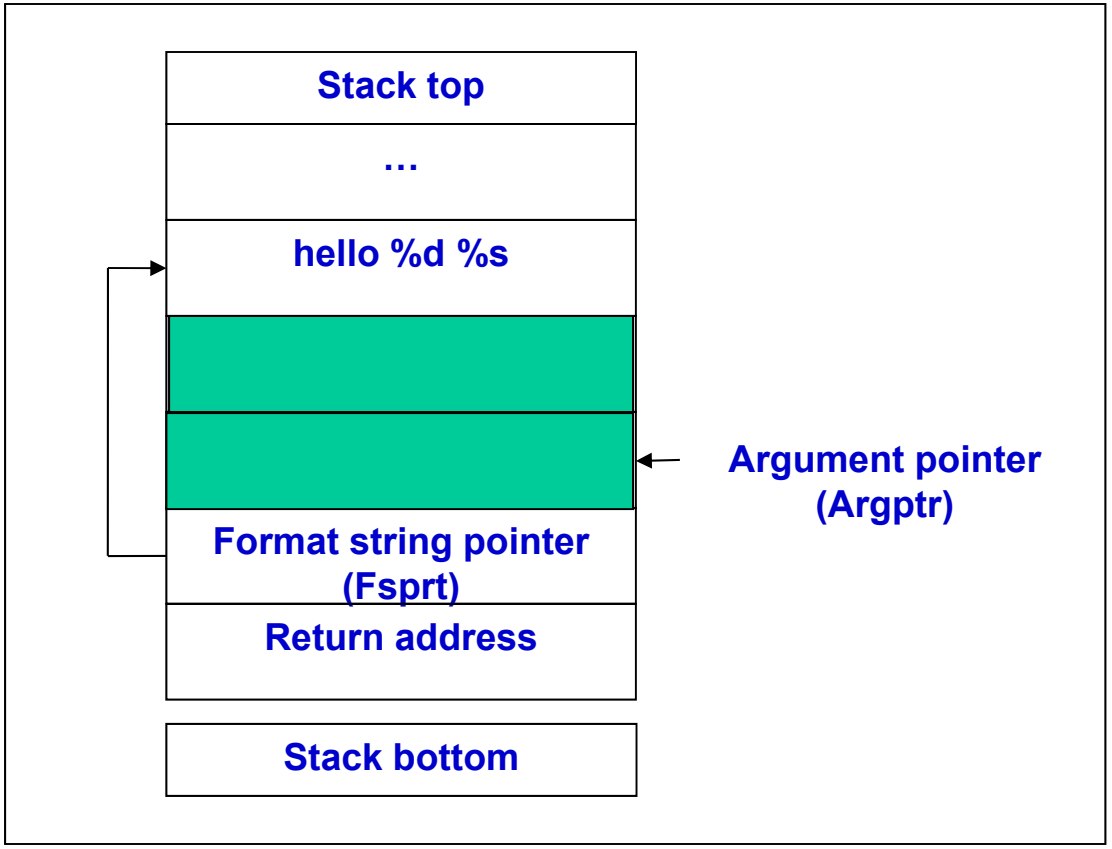
- Invocation of format functions with non-validated user supplied inputs which contain format strings [3]
  - E.g., format functions of ANSI C standard library

- Consequences
  - Arbitrary reading and writing to format function stacks

Format string bug example

printf ("hello %d %s", i, str) [GOOD]

printf ("hello %d %s") [BAD]



Stack of the printf function call (adapted from [3])

SQL Injection (SQLI) vulnerabilities

- SQL queries generated with user supplied inputs that are not properly validated
- Consequences of SQLI vulnerabilities
  - Authentication bypassing
  - Leakage of private information

Example of JSP code with SQLI vulnerability*

```
1. String LoginAction (HttpServletRequest request, ...) throws IOException {
2.    String sLogin = getParam (request, "Login");
3.    String sPassword = getParam (request, "Password");
4.    java.sql.ResultSet rs = null;
5.    String qry = "select member_id, member_level from members where ";
6.    qry = qry  + "member_login ='" + sLogin +  "' and member_password ='"  + sPassword + "'";
7.    java.sql.ResultSet rs = stat.executeQuery (qry);
8.    if  (rs.next ()) { // Login and password passed
9.          session.setAttribute ("UserID", rs.getString(1));
           …
      }
```

' or 1=1 --

blank

select member_id, member_level from members
where member_login ='' or 1=1 --' and member_password =''

Tautology attack (arbitrary condition or true = true)

*A JSP code snippet adapted from Online BookStore [6]

SQLI attack examples

- Some common SQLI attacks (Halfond et al. 2006)
- Tautologies
  - ' or 1=1 --
  - 'greg' like '%gr%'
- UNION queries
  - ' union select 1,1 --
- Piggybacked queries
  - '; show tables; --

Cross Site Scripting (XSS) vulnerabilities

- The generation of HTML contents with invalidated inputs
- Inputs are interpreted by browsers as valid HTML tags
  - The intended behavior of generated web pages alters
  - Visible symptoms (pop-up window)
  - Invisible symptoms (reading form fields, cookies)

XSS attack types

- Stored: Injected code is stored by server programs and used to generate response pages later
- Reflected: Injected code is directly sent back to browsers by server programs
- DOM-based: Injected code is processed by the client-side JavaScript without sanitization

Program security vulnerability -Summary

- Vulnerabilities exist in programs due to lack or weak input validation and logic errors
- Vulnerabilities result in observable abnormal behaviors
  - BOF, FSB, SQLI
- Some vulnerabilities result in unobservable behaviors
  - XSS

Program security testing

- Features of security testing approaches
- Test case generation techniques
- Static analysis, Monitoring, Hybrid analysis

Program security testing

- Why program security testing is ignored?
  - Program testing cost up to 50% of a project (Sommerville 2001)
  - Time to market pressure
  - Lack of in depth knowledge among related professionals on how malicious inputs alter expected program behaviors

- Program security testing is a misunderstood task (McGraw et al. 2004)
  - Is it port scanning and packet analysis?
  - Is it examining firewall policies?

Security testing steps

- Test requirement
  - What aspect of program we want to test?
    - Special classes of bugs that may cause security breaches
  - How these bugs are introduced in programs?
    - Implementation language features
    - Library APIs (ANSI C library)
    - Inputs (network protocol data unit)
- Test coverage
  - Does a test suite reveal specific vulnerabilities?
  - Example coverage of program security testing
    - A test suite should reveal all BOF and SQLI vulnerabilities

Security testing steps (cont.)

- Test cases are generated from program and environment
  - E.g., Source code, library APIs
  - How to define the oracle for each test case?
    - Program states and response messages are used to identify the presence or absence of attacks
- Test cases are run against implementations
  - The presence of vulnerabilities are confirmed by comparing program states or outputs with known malicious states or outputs under attacks

Classification of security testing approaches*

- Source of test case
- Test case input type
- Test case generation method

Classification feature: Source of test case

- Artifacts that are used for generating test cases
- Program artifacts
  - Vulnerable APIs (ANSI C library functions)
  - Runtime states (registers or variable values)
- Inputs
  - Valid files, protocol data units (PDUs)
- Environments
  - File systems, networks, and processors

Classification feature: Test case input type

- It varies based on data types and vulnerabilities
- Data received by programs
  - Exploiting BOF involve generating strings of particular lengths
  - Exposing FSB requires strings containing format specifiers
- Vulnerabilities
  - A stored XSS requires two URLs (storing a malicious script and downloading the script)

Classification feature: Test case generation

- Test case generation method
  - Fault injection
  - Attack signature
  - Mutation analysis
  - Search-based

Test case generation: Fault injection

- One of the most widely used test case generation techniques
  - Suitable for black box-based testing
  - Widely used to discover BOF and FSB vulnerabilities
- The objective
  - Corrupt input data and variables
  - Supply to executable programs
  - Observe unexpected responses to conform vulnerabilities
- Fault injection can be divided into three types based on the target of corruption
  - Input
  - Environment
  - Program state

Test case generation: Fault injection (cont.)

- Fault injection in Input data
  - Malform valid data structure into an invalid one
  - E.g., semantic structure modification (replace the date format "mmddyyyy" to "yyyyddmm")
- Fault injection in environment
  - Modify the attributes of program inputs at different execution stages
  - E.g., during program execution, delete a file or toggle file permissions
- Fault injection in program state
  - Test whether program code can handle malformed states or not
  - E.g., data variables that control program execution flow

Test case generation (cont.)

- Test case generation method
  - Fault injection
  - Attack signature
  - Mutation analysis
  - Search-based

Test case generation: Attack signature

- Widely used in security testing of web-based programs
- Replace some parts of normal inputs with attack signatures
  - Signatures are developed from the experience and vulnerability reports
  - E.g., http://www.xyz.com/login.php?name=john&pwd=secret
  - SQLI: http://www.xyz.com/login.php?name=' or 1=1&pwd=
  - XSS: http://www.xyz.com/login.php?name="><script>...</script>&pwd=
- Two ways to traverse web pages
  - Request and response
  - Use case:

Test case generation: Attack signature (cont.)

- Request and response-based method
  - Requests web pages, input fields are filled with attack inputs, and submitted to websites
  - Vulnerability presence is checked by confirming known response messages under attacks
  - Widely used to discover SQLI and XSS vulnerabilities
- Request and response-based method fails to reach inner logics of programs where vulnerabilities might be present
- Use case-based method can alleviate the limitation
  - Relies on interactive user inputs to perform functionalities
  - Generates a collection of user inputs (or URLs)
  - The inputs are replayed back and malicious inputs are injected at specific points of URLs

Test case generation (cont.)

- Test case generation method
  - Fault injection
  - Attack signature
  - Mutation analysis
  - Search-based

Test case generation: Mutation analysis

- Testing of test cases
  - Assessment of test suite adequacy to reveal faults
- Modifies the source code of programs to create mutants
- A mutant is killed if at least one test case generates different output between an implementation and a mutant
- If no test case can kill a mutant, it is said to be equivalent of the original implementation
- Compute the adequacy with mutation score (MS)
  - (Total # of mutants killed) / (Total # of non-equivalent mutants)

Test case generation: Mutation analysis (cont.)

- Example of mutation analysis
  - Relational operator replacement (ROR)

Original program	**Mutant**
1. int foo (int a, int b){	1. int foo (int a, int b){
2. if (a **>** b)	2. if (a **≥** b) **//ΔROR**
3. return (a – b);	3. return (a – b);
4. else	4. else
5. return (a + b);	5. return (a + b);
}	}

Before killing the mutant: T {(3, 2)}

After killing the mutant: T {(3, 2), (4, 4)}

Test case generation: Mutation analysis (cont.)

- Mutation-based security testing
  - Define mutation operators to inject vulnerabilities
  - Identify mutant killing criteria to generate test cases
- The mutation operators are related to attack type coverage
- The killing criteria specify how to compare outputs between a program and a mutant
  - The end output between an program and a mutant (strong)
  - The intermediate program states between a program and a mutant (weak)
  - At any point between the mutated code and the end of a program (firm)

Test case generation: Mutation analysis (cont.)

Category	Operator	Brief description
Mutating library function calls	S2UCP S2UCT S2UGT S2USN S2UVS	Replace *strncpy* with *strcpy*. Replace *strncat* with *strcat*. Replace *fgets* with *gets*. Replace *snprintf* with *sprintf*. Replace *vsnprintf* with *vsprintf*.
Mutating buffer size arguments	RSSBO	Replace buffer size with destination buffer size plus one.
Mutating format Strings	RFSNS	Replace "*%ns*" format string with "*%s*".
	RFSBO	Replace "*%ns*" format string with "*%ms*", where, *m* = size of the destination buffer plus one.
	RFSBD	Replace "*%ns*" format string with "*%ms*", where, *m* is the size of the destination buffer plus Δ.
	RFSIFS	Replace "*%s*" format string with "*%ns*", where *n* is the size of the destination buffer.
Mutating buffer variable sizes	MBSBO	Increase buffer size by one byte.
Removing statements	RMNLS	Remove null character assignment statement.

Test case generation: Mutation analysis (cont.)

Name	**Killing criteria**
C_1	$ES_P \neq ES_M$ (strong)
C_2	Len(Buf_P) $\leq N$ && Len(Buf_M) $> N$ (weak)
P: The original implementation. *M*: The mutant version. ES_P: The exit status of *P*. ES_M: The exit status of *M*. *Len*(Buf_P): Length of *Buf* in *P*. *Len*(Buf_M): Length of *Buf* in *M*. *N*: Buffer size.	

Test case generation: Mutation analysis (cont.)

Category	Operator	Brief description	Killing criteria
Mutating library function calls	S2UCP S2UCT S2UGT S2USN S2UVS	Replace *strncpy* with *strcpy*. Replace *strncat* with *strcat*. Replace *fgets* with *gets*. Replace *snprintf* with *sprintf*. Replace *vsnprintf* with *vsprintf*.	C_1 or C_2
Mutating buffer size arguments	RSSBO	Replace buffer size with destination buffer size plus one.	C_2
Mutating format strings	RFSNS	Replace "*%ns*" format string with "*%s*".	C_1 or C_2
	RFSBO	Replace "*%ns*" format string with "*%ms*", where, *m* = size of the destination buffer plus one.	C_2
	RFSBD	Replace "*%ns*" format string with "*%ms*", where, *m* is the size of the destination buffer plus Δ.	C_1
	RFSIFS	Replace "*%s*" format string with "*%ns*", where *n* is the size of the destination buffer.	C_1 or C_2
Mutating buffer variable sizes	MBSBO	Increase buffer size by one byte.	C_1
Removing statements	RMNLS	Remove null character assignment statement.	C_1 or C_2

Test case generation: Mutation analysis (cont.)

- An example of mutation analysis for BOF vulnerability

String length (src)	**Original program (P)**	**Mutated program (M)**	**Output (P)**	**Output (M)**	**Status**
[32]	char dest [32]; ….. sscanf (src, "%32s", dest);	char dest [32]; ….. sscanf (src, "%40s", dest); //**ΔRFSBD**	No crash	No crash	Live
[256]	As above	As above	No Crash	Crash	Killed

Test case generation: Mutation analysis (cont.)

- Relationship between BOF attack type and mutation operator

Attack	**Operator**
Overwriting return addresses	All except RSSBO and RFSBO
Overwriting stack and heap	All the proposed operators except RMNLS
One byte overflow	RSSBO and RFSBO
Arbitrary reading of stack	RMNLS

Test case generation (cont.)

- Test case generation method
  - Fault injection
  - Attack signature
  - Mutation analysis
  - Search-based

Test case generation: Search-based

- Program test input space might be huge
  - Finding good test cases becomes time consuming
  - E.g., discovering BOF requires generating a large sized strings
- Search algorithms can be applied to generate test cases
  - A random input is chosen as the initial solution
  - The solution is evolved unless an objective function (fitness function) value remains unchanged
  - Fitness functions are designed in ways so that good test cases obtain higher values than that of bad test cases
- Genetic algorithm
  - 1. Generate initial population
  - 2. Assess fitness of each population
  - 3. Select a subset of population to perform crossover and mutation
  - 4. Repeat steps 2 and 3 unless maximum iteration is reached or fitness value of population remains unchanged

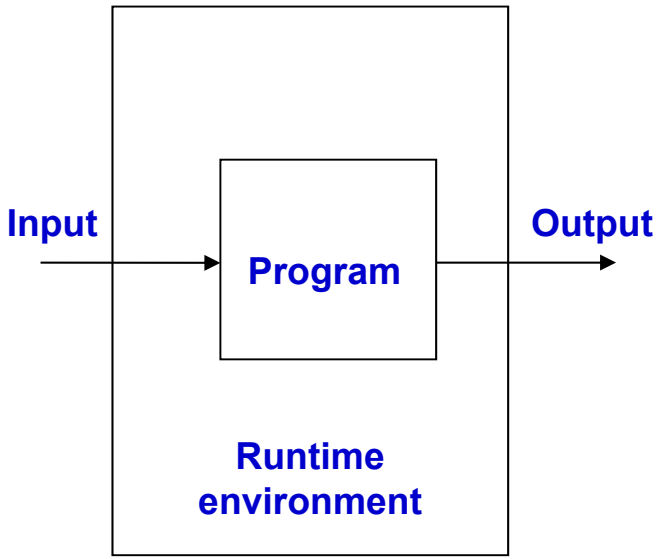
Test case generation: Search-based (cont.)

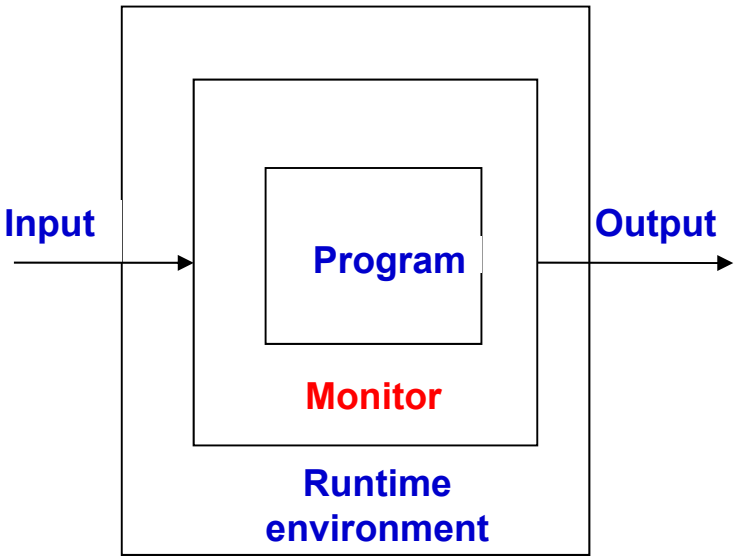
- Fitness function
  - Assess generated test cases based on a numeric score
  - Guides the choosing of test cases to reveal vulnerabilities
- Mutation operator
  - Modification of some random elements in the population to allow evolution of solution

Static analysis

- Identify potential vulnerabilities by analyzing source code
- Static analysis techniques do not execute program code
  - Ignore related issues such as environment variables and test cases for reaching specific program points
- Types
  - Tainted data flow-based
  - Constraint-based
  - Annotation-based
  - String pattern matching-based

A generic program monitor





A program without monitor

Program takes input, process them, and generates output

A program with monitor

A monitor adds an extra layer between program and execution environment

Monitoring aspect

- Program runtime properties that are checked
  - Program operation
  - Code execution flow and origin
  - Code structure
  - Value integrity
  - Unwanted value
  - Invariant

Hybrid analysis

- Static analysis suffers from false positive warnings
  - Programmers spend significant time for examining warnings
- The combination of static and dynamic analysis (hybrid analysis, Ernst et al. 2003)
  - Static analysis identifies program locations that need to be examined
  - Exploitations of vulnerabilities are confirmed by executing programs with test cases and examining runtime states (dynamic analysis)
    - E.g., runtime value stored in a sensitive program location

Hybrid analysis vs. monitoring

- Similarity
  - Both techniques examine runtime program states
    - States vary such as data structure, memory locations, etc.
- Dissimilarity
  - Dynamic analysis is an active analysis
    - Apply before the release of a program
    - Used in a wide range of applications (profiling, debugging)
  - Monitoring is a passive analysis technique
    - Apply after the release of a program
    - Do not stop a program execution unless there is a deviation of expected program behaviors
    - Used in (attack/intrusion detection)

Program security testing - Summary

- Security testing is analogous to software testing, except some changes in defining requirements, coverage, and oracle
- Fault injection is the most widely used technique
  - Provide no information related to vulnerability and code coverage
- Attack signature-based techniques are mainly applied to web-based programs
- Each mitigation technique is important and offers unique advantages
  - Static analysis, Monitoring, Hybrid analysis

References – 1

[1] M. Dowd, J. McDonald, and J. Schuh, The Art of Software Security Assessment, Addision-Wesley publications, 2007.

[2] Aleph One, "Smashing the Stack for Fun and Profit," Phrack Magazine, Volume 7, Issue 49, November 1996, Accessed from http://insecure.org/stf/smashstack.html.

[3] Scut/team teso, "Exploiting Format String Vulnerabilities," 2001, Accessed from http://doc.bughunter.net/format-string/exploit-fs.html

[4] W. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," Proc. of International Symp. on Secure Software Engineering , Virginia, USA, March 2006.

[5] Cross Site Scripting, http://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[6] Online BookStore Web Application, http://www.gotocode.com/apps.asp?app_id=3

[7] JSP Blog, http://jspblog.sourceforge.net/

References – II

[1] H. Shahriar and M. Zulkernine, "Mitigating Program Security Vulnerabilities: Approaches and Challenges," *ACM Computing Surveys*

[2] H. Shahriar and M. Zulkernine, "Assessing Test Suites for Buffer Overflow Vulnerabilities," International Journal of Software Engineering and Knowledge Engineering (IJSEKE), World Scientific, Vol. 20, Issue 1, February 2010, pp. 73-101.

[3] A. Mathur, *Foundations of Software Testing*, First edition, Pearson Education, 2008.

[4] L. Morell, "A Theory of Fault-based Testing," *IEEE Transactions on Software Engineering*, Volume 16, Issue 8, 1990, pp. 844–857.

[5] B. Potter and G. McGraw, "Software Security Testing," *IEEE Security & Privacy*, 2004, pp. 32-34.

[6] I. Sommerville, *Software Engineering*, Addison Wesley, 2001.