Universität Hamburg

Scanstud
Evaluating Static Analysis Tools

OWASP Europe 2008 / Gent
22.05.2008

Martin Johns, Moritz Jodeit
University of Hamburg, Germany

Wolfgang Koeppl, Martin Wimmer
Siemens CERT, Germany

Fachbereich Informatik
SVS – Sicherheit in Verteilten Systemen

ScanStud: Project overview

Mission statement

- Investigating the state of the art in static analysis

Project overview

- Practical evaluation of commercial static analysis tools for security
- Focus on C and Java
- 09/07 – 02/08
- Joint work of University of Hamburg and Siemens CERT

Agenda

1. Introduction

2. Test methodology

3. Test code

4. Experiences and lessons learned

Agenda

1. **Introduction**
2. Test methodology
3. Test code
4. Experiences and lessons learned

The disappointing slide

What we WON'T tell you:

- The actual outcome of the evaluation
- Even if we wanted, we were not allowed (NDAs and such)

But:

- We do not consider the precise results to be too interesting
  - An evaluation as ours only documents a snapshot
  - and is outdated almost immediately

However:

- We hopefully will give you a general feel in respect to the current capabilities of static analysis

So, what will we tell you

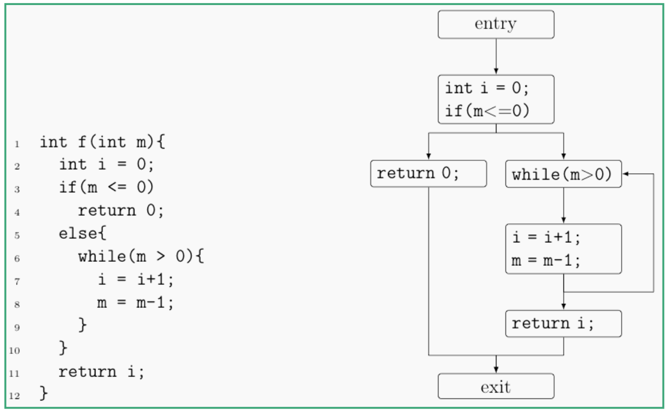
This talk is mainly about our evaluation methodology

- How we did it
- Why we did it this specific way
- General infos on the outcome
- Things we stumbled over

What makes a static analysis tool good?

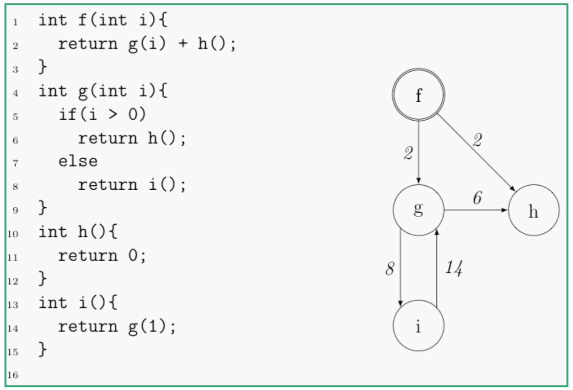
It should find security problems

- Knowledge of different types of code based security problems
    - ◆ E.g., XSS, SQLi, Buffer Overflow, Format String problems...
- Language/Framework coverage
    - ◆ E.g., J2EE servlet semantics, <string.h>,...
- Understanding of flows
    - ◆ Control flow analysis (Loops invariants, integer ranges)
    - ◆ Data flow analysis (pathes from source to sink)



```
int f(int m){
  int i = 0;
  if(m <= 0)
    return 0;
  else{
    while(m > 0){
      i = i+1;
      m = m-1;
    }
  }
  return i;
}
```

Control flow graph



```
int f(int i){
  return g(i) + h();
}
int g(int i){
  if(i > 0)
    return h();
  else
    return i();
}
int h(){
  return 0;
}
int i(){
  return g(1);
}
```

Call graph

Agenda

1. Introduction
2. **Test methodology**
3. Test code
4. Experiences and lessons learned

General approaches towards benchmarking

Approaches

1. Use real world vulnerable software
2. Use existing or selfmade vulnerable application
   - ◆ Hacme, Web Goat, etc...
3. Create specific benchmarking suite

Our goal and how to reach it

- We want to learn a tool’s specific capabilities
  - ◆ E.g., does it understand Arrays? Does it calculate loop invariants? Does it understand inheritance, scoping,...?
- Approaches 1. + 2. are not suitable
  - ◆ Potential side effects
  - ◆ more than one non-trivial operation in every execution path
- Writing custom testcode gives us the control that we need

However the other approaches are valuable too (SAMTE)

Mission Statement

Objectives

- Easy, reliable, correct, and iterative testcase creation
  - → The actual test code should be
    - → short
    - → manual tested
    - → as human readable as possible
- Defined scope of testcases
  - ◆ A single testcase should test only for one specific characteristic
- Automatic test-execution and -evaluation
  - ◆ Allows repeated testing and iterative testcase development
  - ◆ "neutral" evaluation

[Let's start at the bottom]

Automatic test-execution

Approach

- Test-execution via batch-processing

Problem

- All tools behave differently

Solution

- Wrapper applications
  - Unified call interface
  - Unified XML-result format

Automatic test-evaluation

Required

- Reliable mapping between alert and testcode

Approach

- One single vulnerability (or FP) per testcase
- Every testcase is hosted in an application of its own
- The rest of the application should otherwise be clean

Benefits

- Clear relation between alerts and testcases
  - Alert => the case was found / the FP triggerd
  - No alert => the case was missed

Real world problem

Noise

- Even completely clean code can trigger warnings
    - The host-program may cause additional alerts
- How do we deterministically correlate scan-results to test-cases?
    - Line numbers are not always applicable.

Solution

- Result-Diff
    - Given two scan results it extracts the additional alerts
- Scan the host-program only (== the noise)
- Scan the host-program with injected testcase (== signal + noise)
- Diff the results (== signal)

Testcase creation

Approach

- Separation between
  - general support code and
  - test-specific code (the actual vulnerabilities)

Benefit

- Support code is static for all testcases
- The actual testcase-code is reduced to the core of the tested property
  - Minimizes the code, reduces error-rate, increases confidentiality
  - Allows rapid testcase creation
  - Enables clear readability

Implementation

- Code generation
  - Host-program with defined insertion points
  - Testcode is inserted in the host-program

Testcode assembly

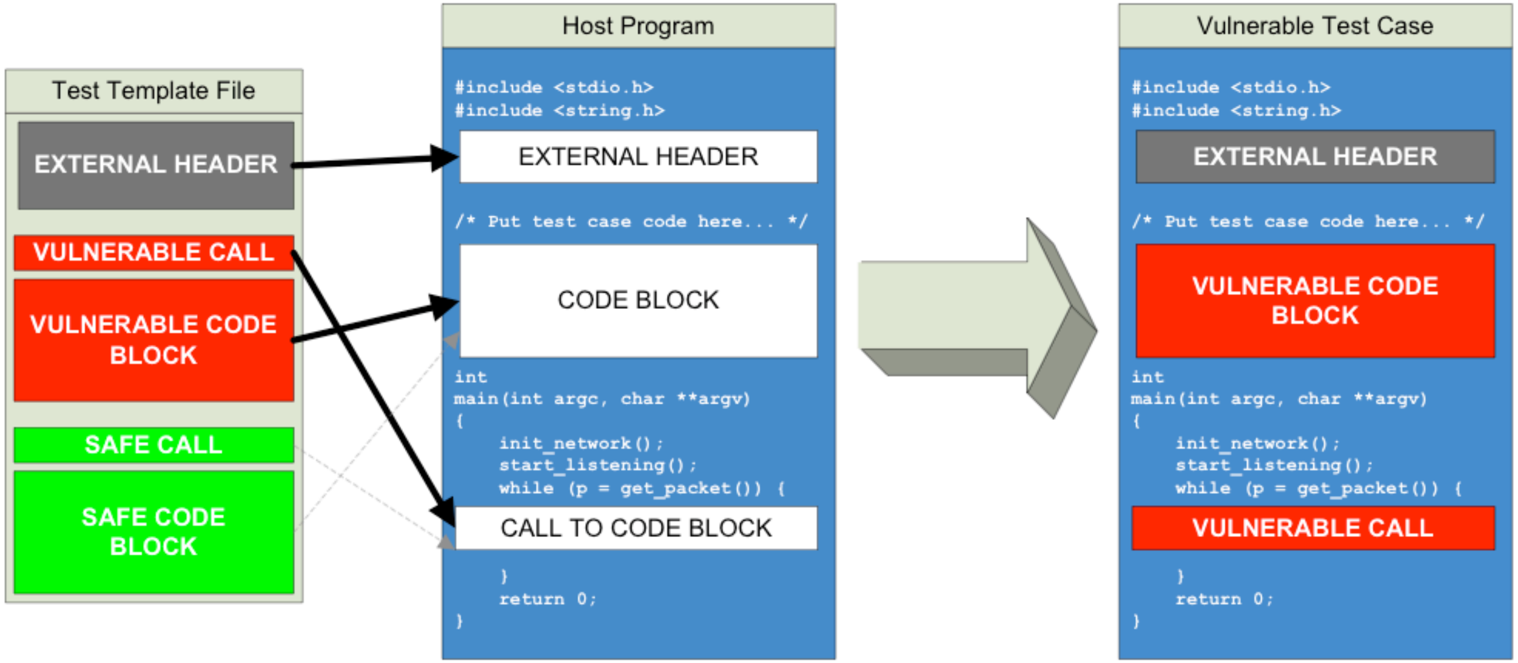
Insertion points in the host program

- Library includes, Global structures/data, function-call to the test function

The test-case is divided in several portions

- Each portion corresponds to one of the insertion points

A script merges the two files into one testcase



Test Template File

EXTERNAL HEADER

VULNERABLE CALL

VULNERABLE CODE BLOCK

SAFE CALL

SAFE CODE BLOCK

Host Program

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

/* Put test case code here... */
CODE BLOCK
int
main(int argc, char **argv)
{
    init_network();
    start_listening();
    while (p = get_packet()) {
CALL TO CODE BLOCK
    }
    return 0;
}
```

Vulnerable Test Case

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

/* Put test case code here... */
VULNERABLE CODE BLOCK
int
main(int argc, char **argv)
{
    init_network();
    start_listening();
    while (p = get_packet()) {
VULNERABLE CALL
    }
    return 0;
}
```

Example testcase(s): Buffer overflow

```
DESCRIPTION:  Simple strcpy() overflow
ANNOTATION:   Buffer Overflow [controlflow] []

EXTERNAL_HEADER:
#include <string.h>

VULNERABLE_CALL: %NAME(v)%(p);

VULNERABLE_EXTERNAL_CODE:
/* %DESCRIPTION(v)% */
void %NAME(v)%(char *p) {
   char buf[1024];
   strcpy(buf, p);      /* %ANNOTATION(v)% */
}

SAFE_CALL: %NAME(s)%(p);

SAFE_EXTERNAL_CODE:
/* %DESCRIPTION(s)% */
void %NAME(s)%(char *p) {
   char buf[1024];
   if (strlen(p) >= sizeof(buf))
       return;
   strcpy(buf, p);      /* %ANNOTATION(s)% */
}
```

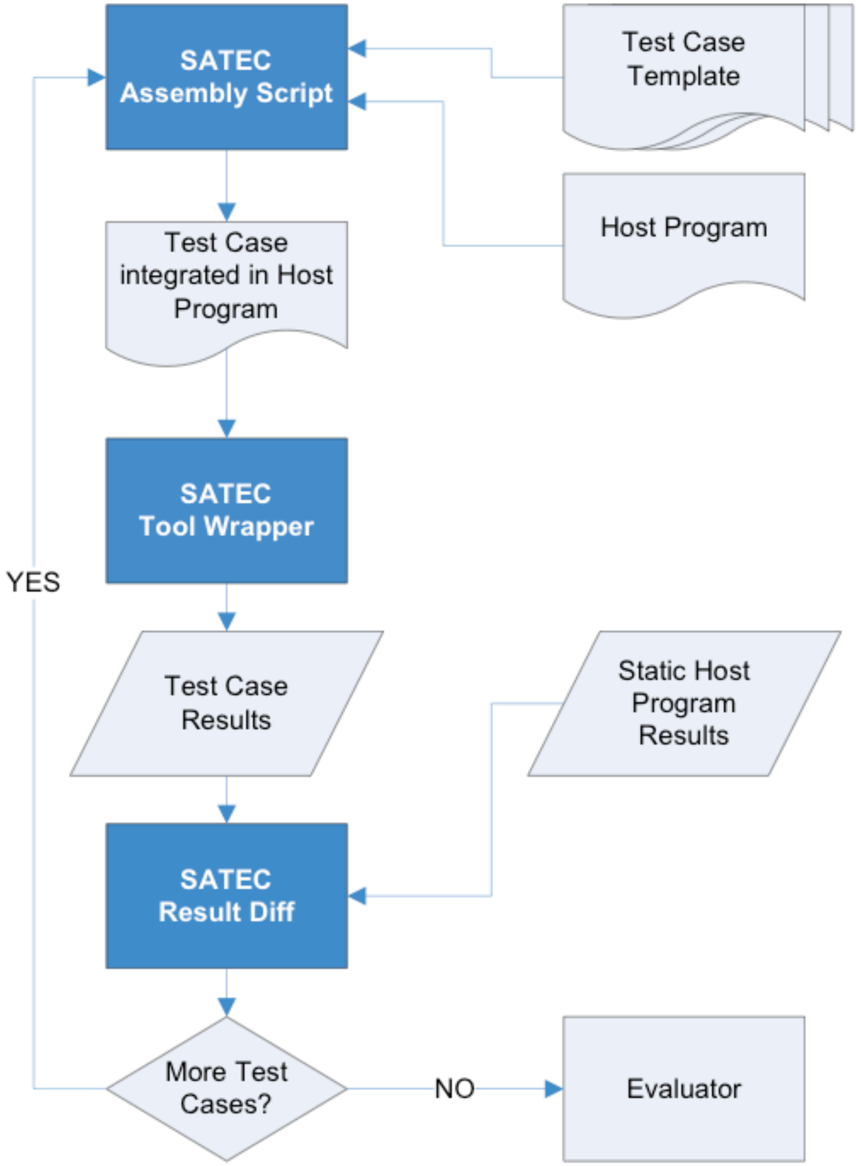
Final testing infrastructure

Components

- Tool wrappers
- Host-program
- Test-cases
- Assembly script
- Result differ
- Evaluator

Putting it all together

- Creates test-code with the assembly-script
- Causes the wrapped tool to access the test-case
- Passes the test-result to result differ
- Diffed-result and meta-data are finally provided to the Evaluator



Conclusion: Test-code generation

Summary

- **Applicable for all potential languages**
- **Applicable for all tools that provide a command-line interface**
- **Flexible**
- **Allows deterministic mapping code <--> findings**

Fallback: Combined suite

- **For cases where the tool cannot be wrapped**
- **All testcases are joined in one big application**

Agenda

1. Introduction
2. Test methodology
3. **Test code**
4. Experiences and lessons learned

Testcases versus Tests

A testcase is the smallest unit in our approach

- Contains code which should probe for exactly one result
- Either “true vulnerability” or “false positive”

A test usually consists of two testcases

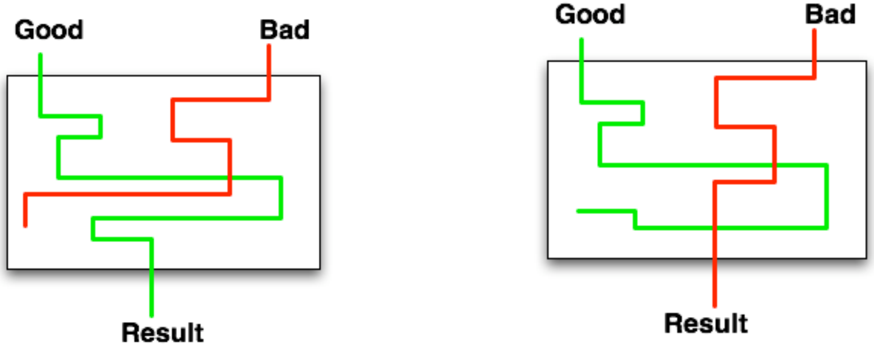
- a true vulnerability and
- a false positive
- Both testing the same characteristic

A test passed only if BOTH associated testcases have been identified correctly

Testcase design

Language features and control/data flow

- Two variables (“good”, “bad”) ⇒ The sources
  - Both are filled with user provided data
  - The “good” variable is properly sanitized
- One sink variable (“result”)
  - This variable is used to execute a security sensitive action
- Both variables are piped through a crafted control flow
- One of them is assigned to the result variable



Memory corruption

- Similar approach
- Instead of variables different sized memory regions are used

C test cases

Host program

- All C test cases are hosted in a simple TCP server
- Listens on a port and waits for new clients
- Reads data from socket and passes pointer to test case
- Less than 100 LOC

The suite

- Emphasis on vulnerability types
- Around 116 single C test cases in total

Tests for, e.g.,

- Buffer overflows, unlimited/Off-by-one pointer loop overflows, integer overflows/underflows, signedness bugs, NULL pointer dereferences

The Java suite

Host program

- J2EE application with only one servlet
  - ◆ Provides: DB connection, framing HTML content, sanitizing,...

Vulnerability classes

- XSS, SQLi, Code Injection, Path Traversal, Response Splitting
  - ⇒ Emphasis on testing dataflow capabilities
- ~ 85 Java testcases in total
  - ◆ Ben Livshit's Stanford SecuriBench Micro was very helpful

Language features

- Library, inheritance, scoping, reflection, session storage

Tests

- Global buffers, array semantics, boolean logic, second order code injection, ...

Agenda

1. Introduction
2. Test methodology
3. Test code
4. **Experiences and lessons learned**

Tool selection

Market research: 12 potential candidates

- Selection criteria:
  - Maturity
  - Is security a core-competence of the tool?
  - Language support

⇒ Selection of 10 tools

⇒ After pre-tests 6 tools were chosen for further investigation

- (no, we can’t tell you which)

Scoring

We have ~ 200 unique testcases

- How should the results be counted?

Observation

- If it aids the detection reliability, false positives are tolerable

Resulting quantification of the results

Test passed:	3 Points
False positive:	1 Point
False negative:	0 Points

Result overview

C Suite

Rank	Tool	Points
1.	Tool a.	72 / 168
2.	Tool b.	58 / 168
3.	Tool c.	56 / 168
4.	Tool d.	53 / 168
5.	Tool e.	50 / 168

Java Suite

Rank	Tool	Points
1.	Tool x.	89 / 147
2.	Tool y.	66 / 147
3.	Tool z.	58 / 147
4.	Tool v.	53 / 147

Static analysis: C capabilities

Categories covered by almost all tools:

- NULL pointer dereferences
- Double free's

Problem areas of most tools:

- Integer related bugs
  - Integer underflows / overflows leading to buffer overflows
  - Sign extension bugs
- Race conditions
  - Signals
  - setjmp() / longjmp()
- Non-implementation bugs
  - Authentication, Crypto, Privilege management, Truncation, …

Static analysis: Java Capabilities

Strengths

- Within a function all tools possess good capabilities to track dataflows
- Besides that, the behaviour/capabilities are rather heterogeneous

Problem areas of most tools

- Global buffers
  - Especially if they are contained within a custom class
- Dataflow in and out of custom objects
  - E.g., our own linked list was too difficult for all tools

```
class Node {
    public    String value;
    public    Node   next;
}
```

- Second order code injection

Static analysis: Anecdotes

Buffer overflows 101:

- **Most basic buffer overflow case?**

```
strcpy()
```

- **To our surprise, 3 out of 5 tools didn’t report this!**
  - **Too obvious to report?**
- **One vendor was provided with this sample:**

```
int main(int argc, char **argv) {
   char buf[16];
   strcpy(buf, argv[1])
}
```

- **Vendor response:**

  “argc/argv are not *modeled* to contain anything sensible.

  We will eventually change that in the future.”

Static analysis: Anecdotes

Buffer overflows 101:

- Another easy one:

```
gets(buf);
```

- Every tool must be finding that one!
  - Actually one tool didn’t
- Vendor response:

“Ooops, this is a bug in our tool.”

Static analysis: Anecdotes

More bugs:

- **One tool didn’t find anything in our “combined test case”:**

```
#include "testcase1.c"
#include "testcase2.c"
#include "testcase3.c"

int main(int argc, char **argv) {
            call_testcase1();
            call_testcase2();
            call_testcase3();
            return 0;
}
```

- **Vendor response:**

> **“#include’ed files are not analyzed *completely*.**
>
> **Will be fixed in a future version.”**

Fun stuff

Let’s sanitize some integers

- All tools allow the specification of sanitation functions
- So did Tool Y
- However the parameter for this function could only be
  - Int, float, ...
  - But not STRING!

Don’t trust the servlet engine

- The J2EE host program writes some static HTML to the servlet response

```
PrintWriter writer = resp.getWriter();
writer.println("<h3>ScanStud</h3>");
```

- Tool X warned “Validation needed”
  - (are you really sure you want your data there?)

More fun and bugs

One of the tools did not find a single XSS problem

- **This surprised us, as the tool otherwise showed decent results**
- **Reason: We used the following code**

```
PrintWriter writer = resp.getWriter();
```

- **But the tool did not know “**`getWriter()`**”**
- **After replacing it with “**`getOutputStream()`**” XSS was found**

Somewhat overeager

- **Our SQLi tests exclusively used SELECT statements**
- **While detecting the vulnerability, the tool Z also warned**
  “stored XSS vulnerability”

A special price: The noisiest tool

We had a tool in round one that did not understood neither C nor Java

- Therefore we started a C# benchmarking suite
- After three written testcases we did a first check
  - 2 XSS (vulnerable/safe), 1 SQLi (vulnerable)

484 Vulnerabilities!

- The tool was not included in the second evaluation round

The end

Questions?

The testing-framework and -code will be published on the SANS website

- **Drop me a line, if you want to be notified (johns@informatik.uni-hamburg.de)**

Appendix

Potential pitfall

Pitfall

- Unbalanced creation/selection of testcases can introduce unsound results

Example

- Tool X is great but does not understand language feature Y
- Therefore all tests involving Y fail
- If there is an unbalanced amount of tests involving Y tool X has an unfair disadvantage

Solution: Categories and tags

- Categories: “controlflow”, “dataflow”, “language”,...
- Tags: All significant techniques within the testcase
  - Example: [cookies,conditional,loops]
- The it would be possible to see, that X allways fails with Y

Interesting point

Vendor X:

- When there is a single path which includes an Array into a vulnerable data-flow, then the whole Array is tainted (even the safe values)
  - Underlying assumption: All elements of a linear data structure are on the same semantic level
  - This approach obviously breaks our test, to examine wether a tool understands Array semantics

C suite

Host program

- All C test cases are hosted in a simple TCP server
- Listens on a port and waits for new clients
- Accepts client connections
- Reads data from socket and passes pointer to test case
- Less than 100 LOC

Test cases

- Around 116 single C test cases in total
- 10 tests to determine the general *performance* of each tool
  - Arrays, loop constructs, structures, pointers, …
- Rest of the test cases represent *real* vulnerabilities, which could be found in the wild

C suite (2)

- Buffer overflows using simple unbounded string functions
  - strcpy, strcat, gets, fgets, sprintf, strvis, sscanf
- Buffer overflows using bounded string functions
  - snprintf, strncpy, strncat, memcpy
- Unlimited/Off-by-one pointer loop overflows
- Integer related bugs
  - Integer overflows / underflows
  - Sign extension
- Race conditions
  - Signals
  - setjmp()
  - TOCTTOU

C suite (3)

- C operator misuse
  - sizeof(), assignment operator, octal numbers
- Format string issues
- NULL pointer derefs
- Memory management
  - Memory leaks
  - Double free’s
- Privilege management
- Command injection
  - popen(), system()

SATEC – Test files

The SATEC file format

- Each test is kept in a separate file
- The test is described using the following keywords
  - NAME (automatically generated from filename)
  - DESCRIPTION
  - ANNOTATION
- Two code blocks
  - VULNERABLE_EXTERNAL_CODE
  - SAFE_EXTERNAL_CODE
- Two calls, into the code blocks
  - VULNERABLE_CALL
  - SAFE_CALL
- Keyword expansion is possible

Example: T_001_C_XSS.java

```
DESCRIPTION:       Very basic XSS
ANNOTATION:        XSS [basic] []

VULNERABLE_CALL:
                   new %NAME(v)%().doTest(req, resp); // inserted by satec

SAFE_CALL:
         new %NAME(s)%().doTest(req, resp); // inserted by satec

VULNERABLE_EXTERNAL_CODE:
class %NAME(v)% extends scanstudTestcase {

         public void doTest(HttpServletRequest req, HttpServletResponse resp){

                   PrintWriter writer = resp.getWriter();
                   String value = req.getParameter("testpar");
                   writer.println("<h3>" + value + "</h3>");   // %ANNOTATION(v)%
         }

}

SAFE_EXTERNAL_CODE:
class %NAME(s)% extends scanstudTestcase {

         public void doTest(HttpServletRequest req, HttpServletResponse resp){

                   PrintWriter writer = resp.getWriter();
                   String value = HTMLEncode(req.getParameter("testpar"));
                   writer.println("<h3>" + value + "</h3>"); // %ANNOTATION(s)%
         }
}
```