

# PHÁT HIỆN LỖI VÀ LỖ HỔNG PHẦN MỀM

## Bài 9. Phát hiện lỗi hổng phần mềm

# Tài liệu tham khảo

1. Đặng Vũ Sơn, Vũ Đình Thu, "Phát hiện lỗi và lỗ hổng phần mềm", Hv KTMM, 2013
2. Freitez, et al. "Software vulnerabilities, prevention and detection methods: a review." *SEC-MDA 2009: Security in Model Driven Architecture*. 2009.
3. Michael Howard, et al., "24 Deadly Sins of Software Security", Mc Graw Hill, 2009
4. Brian Chess, Jacob West, "Secure Programming with Static Analysis", Addison-Wesley, 2007
5. Mark Dowd, et al., "The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities", Addison Wesley, 2006

1

Phân loại chung

2

Phân tích tĩnh

3

Phân tích động

1

Phân loại chung

2

Phân tích tĩnh

3

Phân tích động

# Phân loại

## Sự hiểu biết về phần mềm

- Hộp trắng
- Hộp đen/xám

## Việc thực thi phần mềm

- Tĩnh
- Động

## Sự hỗ trợ của công cụ

- Thủ công
- Bán tự động

# Sự hiểu biết về phần mềm

Source only

Binary only

Both source and binary

Checked build

Strict black box

# Sự hiểu biết về phần mềm

Source only

Binary only

Both source and binary

Checked build

Strict black box

- Chỉ có mã nguồn
- Nhưng không đầy đủ
- Chỉ có thể phân tích tĩnh
- Áp dụng: hợp đồng phân tích mã nguồn phần mềm

# Sự hiểu biết về phần mềm

Source only

Checked build

Binary only

Strict black box

Both source and binary

- Chỉ có mã máy
- Phân tích động hoặc dịch ngược
- Áp dụng: tìm kiếm lỗ hổng các phần mềm thương mại mã đóng



# Sự hiểu biết về phần mềm

Source only

Checked build

Binary only

Strict black box

Both source and binary

- Có cả mã nguồn và cả mã máy (chương trình đã được biên dịch và hoạt động được)
- Áp dụng:
  - Thường chỉ áp dụng khi phân tích nội bộ
  - Hoặc phân tích theo hợp đồng khi có yêu cầu cao về tính an toàn

# Sự hiểu biết về phần mềm

Source only

Binary only

Both source and binary

Checked build

Strict black box

- Chỉ có mã máy
- Nhưng được build ở chế độ debug
- Áp dụng: khi không muốn cung cấp mã nguồn nhưng muốn người phân tích có thêm thông tin về phần mềm

# Sự hiểu biết về phần mềm

Source only

Checked build

Binary only

Strict black box

Both source and binary

- Không có mã nguồn, cũng không có mã máy
- Chỉ có giao diện để giao tiếp với phần mềm
- Chỉ có thể phân tích hộp đen kiểu fuzzing
- Áp dụng: thường được sử dụng để phân tích ứng dụng web

1

Phân loại chung

2

Phân tích tĩnh

3

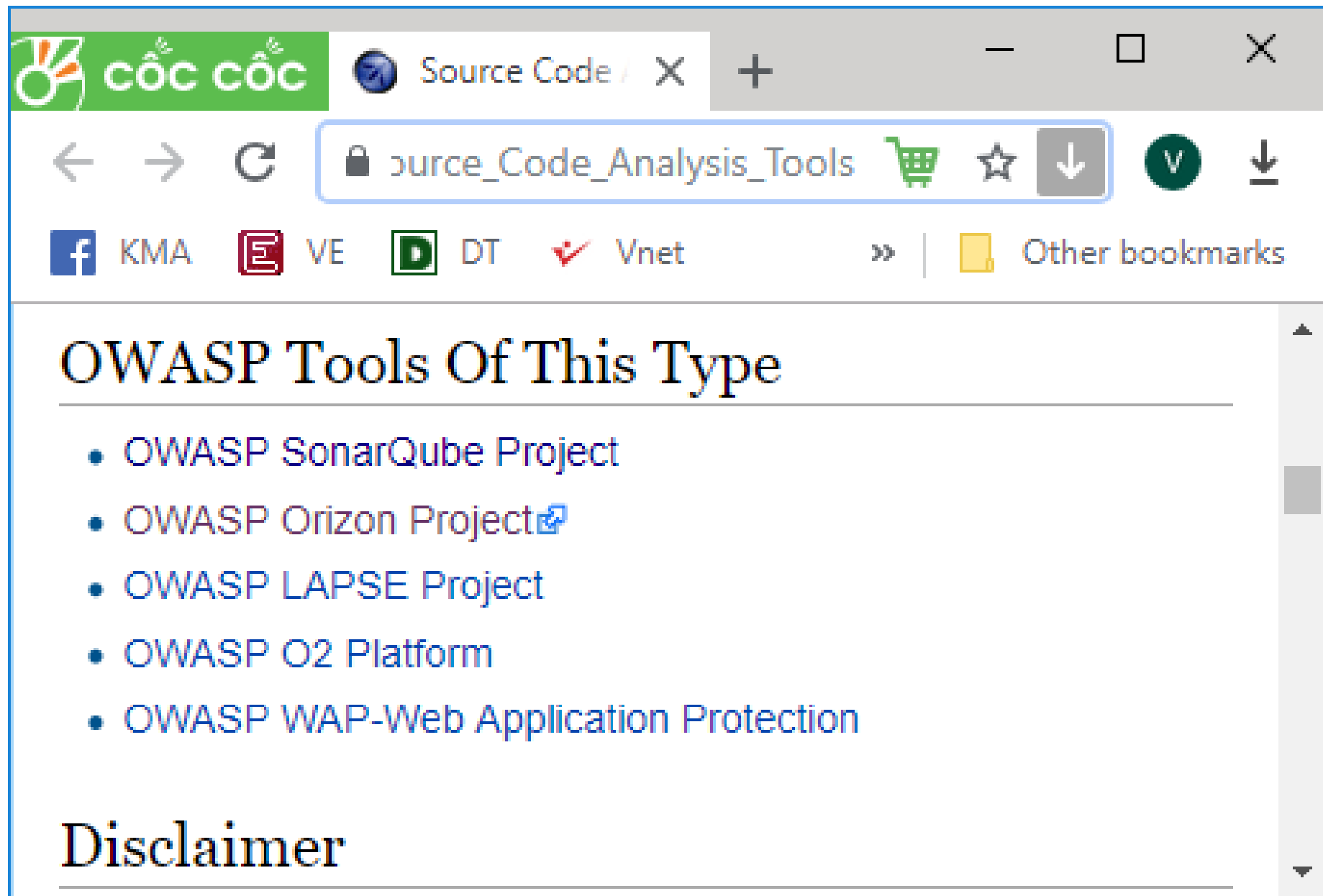
Phân tích động

# Phân tích tĩnh

---

- SAST = Static Application Security Testing
- Source Code Analysis Tools
- Source Code Security Analyzers

# Phân tích tĩnh



The screenshot shows a web browser window with the following elements:

- Tab:** Source Code
- Address Bar:** source\_Code\_Analysis\_Tools
- Bookmarks:** KMA, VE, DT, Vnet, Other bookmarks
- Page Title:** OWASP Tools Of This Type
- Content:**
  - OWASP SonarQube Project
  - OWASP Orizon Project
  - OWASP LAPSE Project
  - OWASP O2 Platform
  - OWASP WAP-Web Application Protection
- Footer:** Disclaimer

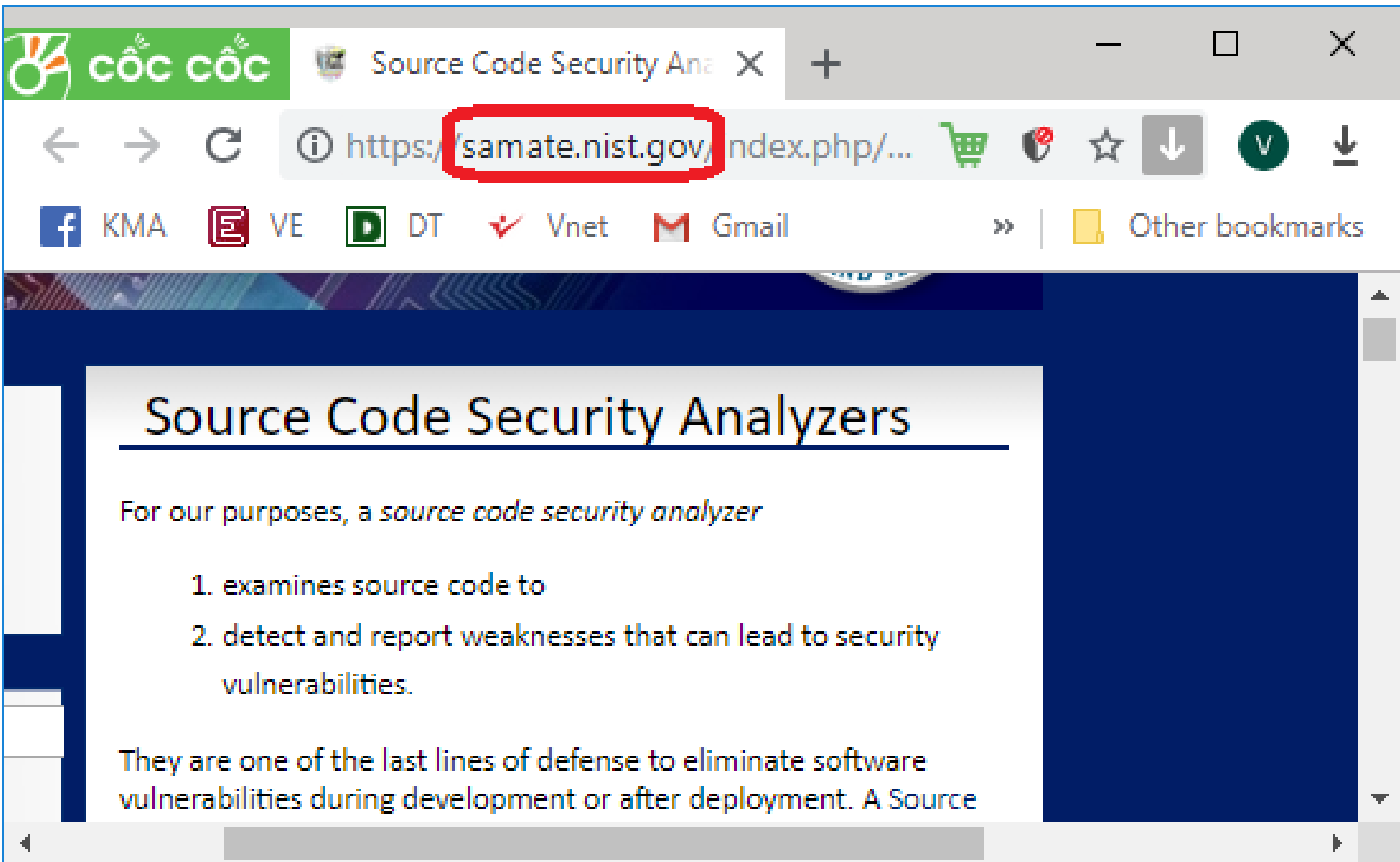
# Phân tích tĩnh



The screenshot shows a web browser window with the following elements:

- Browser Tab:** Labeled "Source Code" with a close button (X).
- Address Bar:** Contains the URL "source\_Code\_Analysis\_Tools". It includes navigation icons (back, forward, refresh), a lock icon, a shopping cart icon, a star icon, a download icon, and a circular icon with a 'V'.
- Bookmarks Bar:** Displays several bookmarks: "KMA" (Facebook icon), "VE" (document icon), "DT" (document icon), "Vnet" (checkmark icon), and "Other bookmarks" (yellow folder icon).
- Page Content:**
  - ## Open Source or Free Tools Of This Type
  - [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

# Phân tích tĩnh





# Phân tích tĩnh

---

1. Pattern Matching
2. Lexical Analysis
3. Parsing
4. Taint Analysis

# Phân tích tĩnh

- Đặc điểm chung của phân tích tĩnh là tỷ lệ cảnh báo nhầm cao
- Chủ yếu là phân tích hộp trắng
- Người ta kết hợp với kỹ thuật học máy (machine learning) để giảm thiểu cảnh báo nhầm
- Một số giải pháp cho phép tự động khắc phục lỗi/lỗi hổng khi được phát hiện

## Pattern matching

- Tìm kiếm mẫu trong mã nguồn
- Ví dụ mẫu: sử dụng các hàm nguy hiểm như `gets()`, `printf()`
- Nhiều cảnh báo nhầm
- Khó xây dựng các mẫu phức tạp để giảm cảnh báo nhầm, bởi chỉ thêm một dấu cách cũng khiến mẫu không khớp

## Lexical analysis

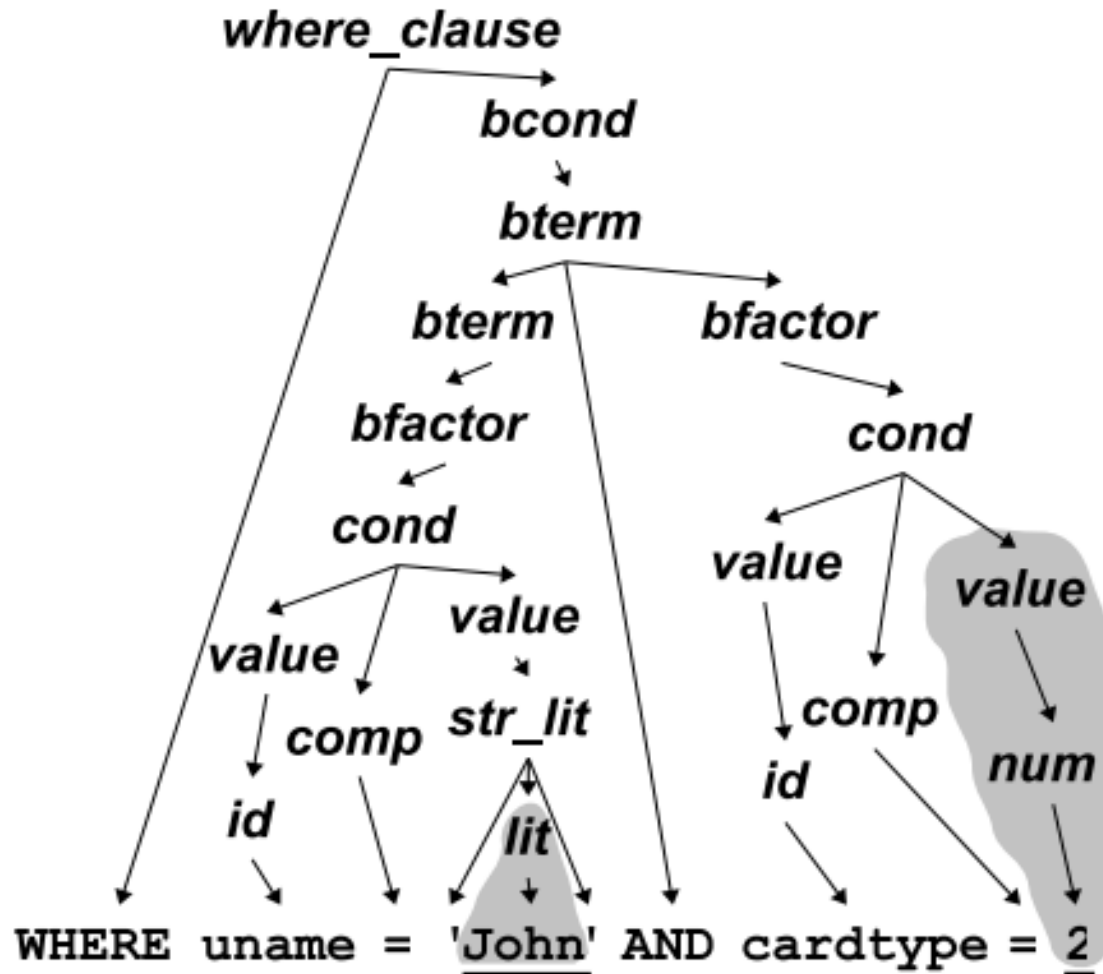
- Là phân tích mã thành từ vựng
- Thực hiện trước khi áp dụng Pattern Matching. Khắc phục khó khăn khi xây dựng mẫu

## **Parsing (Syntax Analysis)**

- Là phân tích cú pháp của mã nguồn
- Thực hiện sau Lexical analysis
- Xây dựng cấu trúc dạng cây để xác định ngữ nghĩa của mã nguồn

# Phân tích tĩnh

- Kết quả parsing một mệnh đề WHERE, chỉ ra dữ liệu do người dùng nhập vào



## Taint analysis

– Đánh dấu sự lan truyền của dữ liệu do người dùng nhập vào

```
1: $a = $_GET['user'];
2: $b = $_POST['pass'];
3: $c = "SELECT *FROM users WHERE
u='mysql_real_escape_string($a)';
4: $b = "wap";
5: $d = "SELECT *FROM users WHERE u= '$b'";
6: $r = mysql_query($c);
7: $r = mysql_query($d);
8: $b = $_POST['pass'];
9: $query = "SELECT *FROM users WHERE u= '$a' AND p='$b'";
10: $r = mysql_query($query);
```

1

Phân loại chung

2

Phân tích tĩnh

3

Phân tích động



# Phân tích động

---

1. Fault Injection
2. Fuzzing Testing

## ❑ Fault Injection

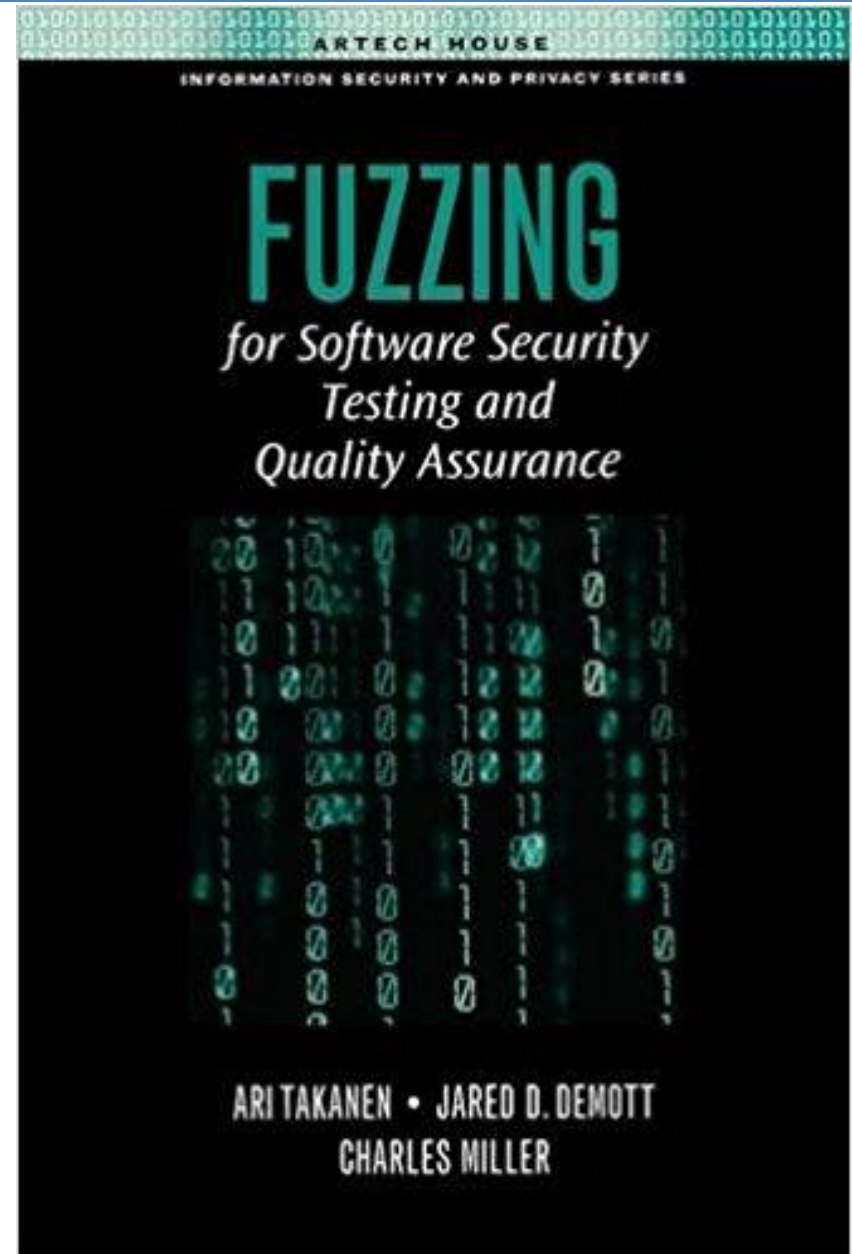
- "Tiêm lỗi": đưa vào các dữ liệu khiến phát sinh lỗi thực thi
- Có lỗi phát sinh → có khả năng có lỗi hổng  
→ phân tích sâu hơn để kiểm chứng
- Cần có thông tin về chương trình (hộp trắng)

## □ Fuzzing Testing

- Là trường hợp riêng của Fault Injection
- Mục đích là làm phát sinh lỗi thực thi
- Có lỗi → có thể có lỗ hổng → phân tích sâu hơn để kiểm chứng
- Ưu điểm:
  - Không cần thông tin về chương trình
  - Dữ liệu được sinh tự động

## 8

MICHAEL SUTTON  
ADAM GREENE  
PEDRAM AMINI



# Fuzzing testing

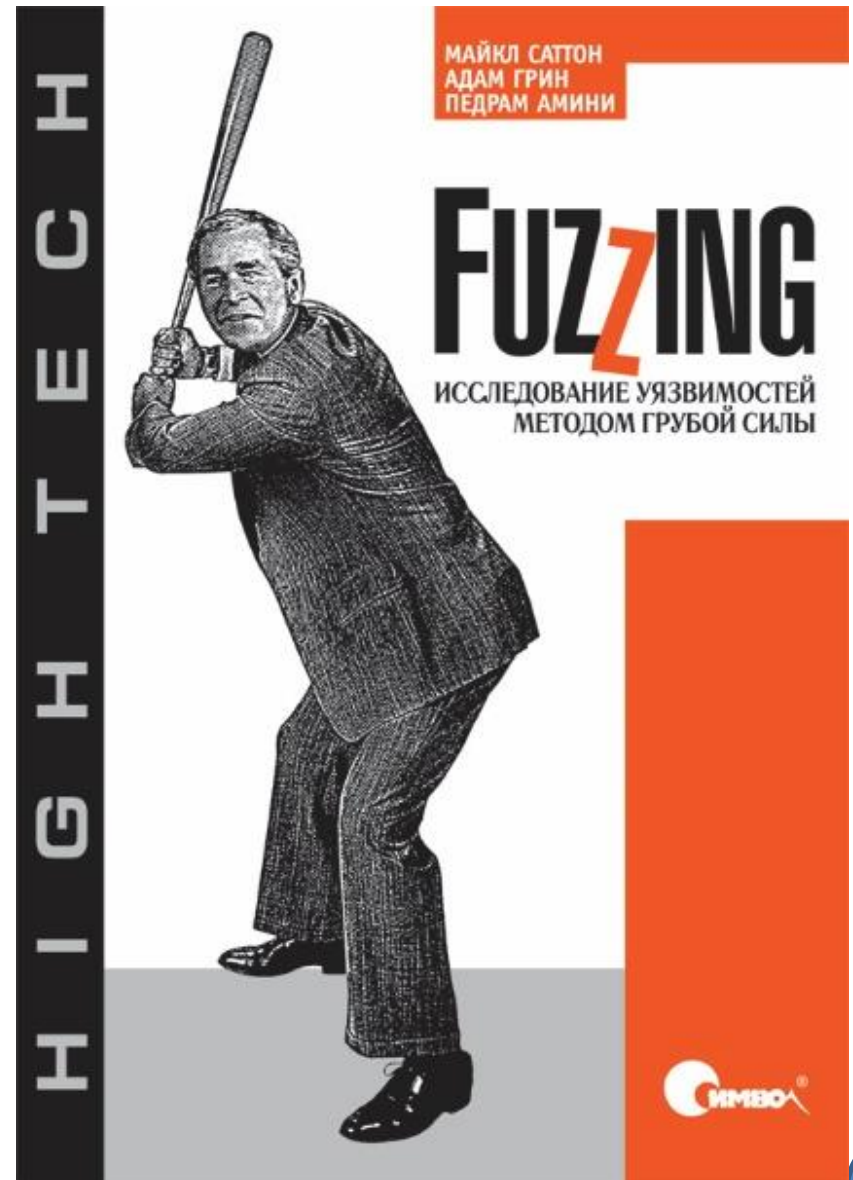
## Fuzzing for software vulnerability discovery

Toby Clarke

Technical Report  
RHUL-MA-2009-04  
17 February 2009

Royal Holloway  
University of London

Department of Mathematics  
Royal Holloway, University of London  
Egham, Surrey TW20 0EX, England  
<http://www.rhul.ac.uk/mathematics/techreports>

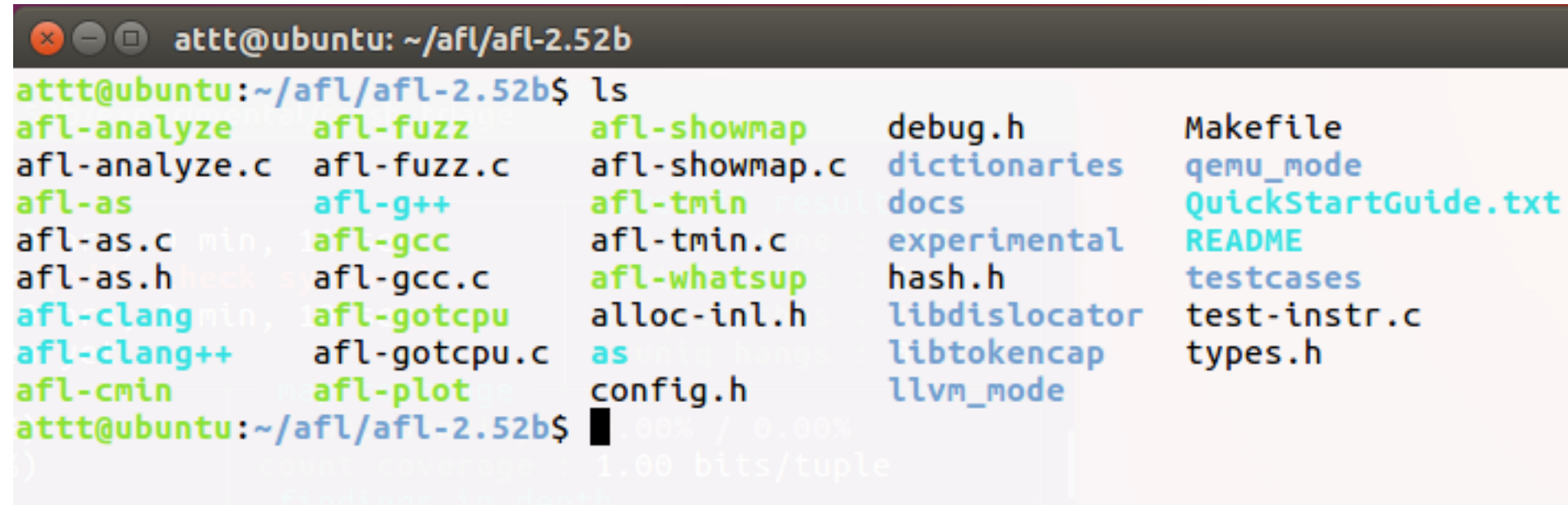


# Fuzzers

- Mã nguồn mở, Thương mại
- Rất nhiều mã nguồn mở!!!!!!!!!!!!!!!!!!!!  
VUzzer, afl-fuzz, filebuster, TriforceAFL,  
Nightmare, grr, Randy, IFuzzer, Dizzy,  
Wfuzz, Go-fuzz, Sulley, Sulley\_l2, CERT  
Basic Fuzzing Framework (BFF), CERT  
Failure Observation Engine (FOE),  
DrangerFor ActiveX Controls, Radamsa...

# AFL – Download and Build

```
$ wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
$ tar -xf afl-latest.tgz
$ cd afl-<version>
$ make
```



```
attt@ubuntu: ~/afl/afl-2.52b
attt@ubuntu:~/afl/afl-2.52b$ ls
afl-analyze      afl-fuzz          afl-showmap      debug.h          Makefile
afl-analyze.c    afl-fuzz.c        afl-showmap.c    dictionaries     qemu_mode
afl-as           afl-g++           afl-tmin         docs             QuickStartGuide.txt
afl-as.c         afl-gcc           afl-tmin.c       experimental     README
afl-as.h         afl-gcc.c         afl-whatsup      hash.h           testcases
afl-clang        afl-gotcpu        alloc-inl.h     libdislocator    test-instr.c
afl-clang++      afl-gotcpu.c      as              libtokencap      types.h
afl-cmin         afl-plot          config.h         llvm_mode
attt@ubuntu:~/afl/afl-2.52b$
```

# AFL – Configure

```
$ alias afl-gcc= path-to-afl/afl-gcc
```

```
$ alias afl-fuzz= path-to-afl/afl-fuzz
```

```
$ alias afl-tmin= path-to-afl/afl-tmin
```

```
$ alias afl-showmap= path-to-afl/afl-showmap
```

//Force the system to make dump files when crash

```
$ sudo sysctl -w kernel.core_pattern = ~/afl/core.%e.%p
```



# AFL – Build the target (with instrumentation)

//For a project

```
$ CC=/path-to-afl/afl-gcc
```

```
$ make clean all
```

//For a program without make

```
$ afl-gcc <main.c> [-o <app>] [other options]
```

# AFL – Prepare and Start

---

```
$ mkdir testcases
```

```
$ mkdir out
```

```
//Create some testcases
```

```
$ echo "hello" > testcases/foo
```

```
//Start fuzzing
```

```
$ afl-fuzz -i testcase -o out -- ./app
```

# AFL - running

```
attt@ubuntu: ~/afl/afl-2.52b/experimental/crash_triage

process timing
  run time : 0 days, 0 hrs, 0 min, 11 sec
  last new path : none yet (odd, check syntax!)
  last uniq crash : 0 days, 0 hrs, 0 min, 10 sec
  last uniq hang : none seen yet

cycle progress
  now processing : 0 (0.00%)
  paths timed out : 0 (0.00%)

stage progress
  now trying : havoc
  stage execs : 79/256 (30.86%)
  total execs : 43.6k
  exec speed : 4003/sec

fuzzing strategy yields
  bit flips : 0/32, 0/31, 0/29
  byte flips : 0/4, 0/3, 0/1
  arithmetics : 0/224, 0/0, 0/0
  known ints : 0/23, 0/84, 0/44
  dictionary : 0/0, 0/0, 0/0
    havoc : 1/43.0k, 0/0
    trim : 33.33%/1, 0.00%

map coverage
  map density : 0.00% / 0.00%
  count coverage : 1.00 bits/tuple

findings in depth
  favored paths : 1 (100.00%)
  new edges on : 1 (100.00%)
  total crashes : 63 (1 unique)
  total tmouts : 0 (0 unique)

path geometry
  levels : 1
  pending : 0
  pend fav : 0
  own finds : 0
  imported : n/a
  stability : 100.00%

overall results
  cycles done : 165
  total paths : 1
  uniq crashes : 1
  uniq hangs : 0

^C [cpu000:103%]
```

