Dynamic Tainting

• **Dynamic** (= Runtime) vs. **Static** (= Compile-time)
  • against unknown, new attacks
    • hard to remove all kinds of vulnerabilities
  • fast reaction via error detection, filtering, etc.
    • spare time for correction, e.g. patch

• **Tainting**
  • taint, trace, and test
    • what to taint
    • how to trace
    • when/where to test
Two applications

<table>
<thead>
<tr>
<th></th>
<th>TaintCheck</th>
<th>DYTAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>what to taint</td>
<td>data from untrusted source, e.g. network, stdin, specific files, etc.</td>
<td>statically/dynamically(^1) allocated memory and pointers</td>
</tr>
<tr>
<td>how to trace</td>
<td>(Taint Data Structure) instrumentation before each data movement or arithmetic instruction</td>
<td>(HW-assisted Taint Mark) extend processor core(^2) to propagate taint marks</td>
</tr>
<tr>
<td>when/where to test</td>
<td>(configurable policies) tainted?: jump target, format strings, or sys call args.</td>
<td>for any memory access, the taint marks of pointer and memory conform?</td>
</tr>
<tr>
<td>what it prevents</td>
<td>overwrite attacks and format string attacks</td>
<td>IMA (illegal memory access): overrun, uninitialized, dangling</td>
</tr>
</tbody>
</table>

1. Their prototype handles heap-allocated memory only.
2. They have both SW- and HW-based implementation.
Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software

James Newsome and Dawn Song, NDSS ’05
Motivation: fast-spreading worm

- fast-spreading worms
  - that can infect thousands of hosts in minutes
  - e.g., CodeRed and Slammer
- attacks for previously unknown vulnerabilities
- not enough w/ coarse-grained detectors
  - that detect malicious behavior
    - e.g., scanning or listening to ports
  - due to frequent false positives and negatives
- commodity software
  - unavailable source codes and recompiled binaries
Goal

• automatic
  • all processes including detection, analysis, and attack signature generation
• configurable
  • easy to deploy due to generalized components
  • open rooms for policy modification
• fine-grained
  • resulting in fewer false positives and false negatives
  • providing detailed information about the attack
• binary-level
Design

- mark input from untrusted source as “tainted”
- track how tainted data propagate by monitoring execution
- check whether tainted data is used in dangerous way
• based on Valgrind x86 emulator
• Valgrind translates x86 code to RISC-like UCode
• TaintCheck codes are instrumented
• translated back to x86 executable by Valgrind
TaintSeed

- marks data coming from untrusted sources
  - e.g., network sockets, stdin, and file systems
- for each byte, 4-byte shadow memory that points to the Taint Data Structure
TaintTracker

- traces the propagation of tainted data
  - data movement: taint dest iff any byte of src is tainted
  - arithmetic op.: taint result iff any byte of operands is tainted
- if the result is tainted, set shadow memory to point to the same TDS (or allocate a new one w/ the link to the source)
TaintAssert

- checks whether tainted data is unwisely trusted
  - policies can be customized
- By default, TaintAssert checks uses of tainted data as:
  - jump addresses, format strings, system call arguments, and application or library-specific checks

```
printf(buf);
```
Exploit Analyzer

• gathers the information regarding how the exploit happened
  • backtracking the program execution path in between tainted
data’s entry and its use in an exploit
  • using the logged info. by TaintSeed and TaintTracker, e.g., T.D.S.

• helps to find the nature and location of the vulnerability
  quickly

• helps generate an attack signature for the worm
Security Analysis

- attacks detected by TaintCheck
  - overwrite attack
    - jump targets: return addr., function pt., function pt. offset
  - format string attack
    - attackers can trick the program into leaking data or into writing an attacker-chosen value to attacker-chosen addresses
- false negatives
  - implicit flow by control flow
    - e.g., if (x == 0) y = 0; else if (x == 1) y = 1; ...
  - used as an index into a table
  - data from overconfident resources
- false positives
  - either the presence of vulnerability or sanity-checked operations
Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>Overwrite Method</th>
<th>Overwrite Target</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATPhttpd</td>
<td>buffer overflow</td>
<td>return address</td>
<td>✔️</td>
</tr>
<tr>
<td>synthetic</td>
<td>buffer overflow</td>
<td>function pointer</td>
<td>✔️</td>
</tr>
<tr>
<td>synthetic</td>
<td>buffer overflow</td>
<td>format string</td>
<td>✔️</td>
</tr>
<tr>
<td>synthetic</td>
<td>format string</td>
<td>none (info leak)</td>
<td>✔️</td>
</tr>
<tr>
<td>cfingerd</td>
<td>syslog format string</td>
<td>GOT entry</td>
<td>✔️</td>
</tr>
<tr>
<td>wu-ftp</td>
<td>vsnprintf format string</td>
<td>return address</td>
<td>✔️</td>
</tr>
</tbody>
</table>

- synthetic exploits
  - overwritten return address
  - overwritten function pointer
  - format string vulnerability

- actual exploits
  - buffer overflow: ATPhttpd
  - format string vulnerability: cfingerd, wu-ftp
Performance

- CPU-bound: bzip2
  - normal 8.2s v.s. TaintCheck 305s (37.2 times longer)
- Short-lived: cfingerd
  - 36 times as long
- Common case: Apache
  - performance penalty should not be noticeable due to latency caused by network and/or disk I/O
Automatic Signature Generation
Effective Memory Protection Using Dynamic Tainting

James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic, ASE ’07
Motivation: IMA

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for (i = 0; i <= n; i++)
        *(buf + i) = rand();
    ...
}
```
Goal

• Previous work
  • static techniques (language- and analysis-based)
    • require source codes
  • dynamic techniques
    • analysis-based: unacceptable overhead
    • hardware-based: extensive modification

• operate at the binary level

• HW-based approach to reduce overhead

• minimal, practical modifications
Taint Marks

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for (i = 0; i <= n; i++)
        *(buf + i) = rand();

    ...
}
```
Tainting

void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for (i = 0; i <= n; i++)
        *(buf + i) = rand();
    ...
}
Propagating

- taint marks at memory are never propagated, rather cleared when an area of memory is being deallocated.
- taint marks at pointers
  - addition/subtraction, Bitwise NOT
    - most common use of +/- is to +/- a pointer and an offset
    - “numeric” taint marks
      - for given t_a and t_b marks, make t_a +/- t_b taint marks
      - c = ~a : taints c with ¬t_a mark
  - Bitwise AND
    - according to a mask
      - c = a & 0xffffffff00 : base addr., taints with the same taint mark
      - c = a & 0x000000ff : offset, untainted
  - *, /, OR, XOR
    - found zero cases where the result of these op was a pointer
Checking

• No matter when memory is accessed through a pointer: compare the taint marks of the memory and the pointer
• legal ONLY IF they are identical!

<table>
<thead>
<tr>
<th>pointer</th>
<th>memory</th>
<th>IMA?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>yes</td>
</tr>
</tbody>
</table>

void main() {
    int *np, n, i, *buf;

    np = &n;

    printf(“Enter size: “);
    scanf(“%d”, np);

    buf = malloc(n * sizeof(int));

    for (i = 0; i <= n; i++)
        *(buf + i) = rand();

    ...
}

Preventing IMAs
Limiting # of Taint Marks

- ideally, unlimited number of unique taint marks
  => infeasible for HW implementation
  - increases the overhead in terms of time and space (e.g. cache)
  - complicates the design

- assign taint marks from a limited, reusable pool
  - the same taint mark is never assigned to contiguous regions
    - from the observation that most of common program defects occur when a pointer accesses an adjacent memory region occasionally

- although the detection is probabilistic,
  in practice, the approach is successful with only 2 taint marks!
Effectiveness

<table>
<thead>
<tr>
<th>Application</th>
<th>IMA location</th>
<th>Type</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc-1.06</td>
<td>more_arrays: 177</td>
<td>buffer overflow</td>
<td>✓</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>lookup: 177</td>
<td>buffer overflow</td>
<td>✓</td>
</tr>
<tr>
<td>gnupg-1.4.4</td>
<td>parse_comment: 2095</td>
<td>integer overflow</td>
<td>✓</td>
</tr>
<tr>
<td>mutt-1.4.2.li</td>
<td>utf8_to_utf7: 199</td>
<td>buffer overflow</td>
<td>✓</td>
</tr>
<tr>
<td>php-5.2.0</td>
<td>string.c: 3152</td>
<td>integer overflow</td>
<td>✓</td>
</tr>
<tr>
<td>pine-4.44</td>
<td>rfc822_cat: 260</td>
<td>buffer overflow</td>
<td>✓</td>
</tr>
<tr>
<td>squid-2.3</td>
<td>ftpBuildTitleUrl: 1024</td>
<td>buffer overflow</td>
<td>✓</td>
</tr>
</tbody>
</table>

• RQ1: is this approach effective enough when using only a small number of taint marks?

• Software implementation
  • PIN, binary instrumentation tool

• Subject: SPEC CPU2000 benchmark (12 applications)
  • 5 applications with 7 known IMAs
Performance

- RQ3: is the efficiency of this approach sufficient to be applied to deployed software?
- Hardware implementation
  - cycle accurate simulator (SESC)
  - store taint markings in a packed array
References

- Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, James Newsome and Dawn Song, NDSS ’05

- Effective Memory Protection Using Dynamic Tainting, James Clause, et al., ASE ’07

- Slides by the authors and Devendra Salvi (03/22/07)