Deconstructing Hardware Architectures for Security

Michael Dalton   Hari Kannan   Christos Kozyrakis

Computer Systems Laboratory

Stanford University

Computer Security

- Computers are a critical component of national infrastructure
  - Store critical information (credit card #s, SSNs, …)
  - Provide essential services (power, phones, banking, …)
- Computer security flaws are pervasive and dangerous
  - Internet is a hostile environment, programmers make mistakes
  - Results in identity theft, data loss, downtime, IP theft, …
  - Costs billions per year
- Must provide safe, reliable computing
  - Confidentiality
  - Integrity
  - Availability

Why Hardware Support for Security?

- Software should have final say on security policy

- But HW support provides crucial advantages
  - High precision with low overhead
    - HW can examine every instruction executed
    - HW can examine every byte accessed
    - Powerful, fine-grain analysis with little or no runtime cost
  - Compatibility
    - Does not require source code access or recompilation
    - Does not require debugging information or binary instrumentation
    - Works with self-modifying, dynamically generated/loaded code

- Proposed architectures for security
  - <u>Tainting architectures</u> to prevent memory corruption exploits
  - <u>Information leak prevention architectures</u> to prevent information leaks by malicious/vulnerable programs

Outline

- Motivation
- Tainting architectures
  - Memory corruption bugs
  - Tainting architectures summary
  - Flaws: input validation, new format string attacks
- Information leakage prevention architectures
  - RIFLE summary
  - Flaws: implicit information flow
- Conclusions

- Read paper for
  - Details of new format string attack
  - Full discussion on information leak, tainting

Memory Corruption Bugs

- Basic idea
  - Programmer bugs allow overwrite of critical memory regions
  - Can result in complete application compromise
- One of oldest and most damaging security flaws
  - The basis for the Internet worm, NIMDA, Slapper, …
  - Several possible forms
    - Buffer overflows, format string bugs, off-by-ones
- Common issue with type-unsafe languages (C and C++)
  - Unsafe languages do not restrict or check memory accesses

Memory Corruption Example

```
int vuln(char * username){
     char buf[512];
     strcpy(buf,username);
     process_input(buf);
     ...
     return 0;
}
```

- No bounds check on strcpy()
  - If username is longer than 512 bytes, buffer overflow…
  - Strcpy() may overwrite other data structures on stack
- Attack
  - Write arbitrary malicious code on the stack
  - Overwrite return address for vuln() function
  - When vuln() returns, malicious code is run…

Tainting Architectures

- Basic model [Suh’04, Crandall’04, Chen’05]
  - Extend each memory/register byte by one “taint” bit
  - Data from untrusted sources tainted by the OS
  - Taint bits propagate across instructions
  - When tainted data used as pointer, trap to OS
    - E.g., use tainted value as instruction pointer in our example

- Several alternatives
  - Dynamic Information Flow Tracking (DIFT) [Suh’04]
  - Minos [Crandall’04]
    - Similar to DIFT for control-only attacks; not discussed
  - Pointer Taintedness Detection (PTD) [Chen’05]

Validating Untrusted Input

- Applications must be able to validate their input
  - E.g., bounds checking

```
int safe(char * username){
   char buf[512];

   if (strlen(username) >= 512) return -1;
   strcpy(buf,username);
   process_input(buf);
   return 0;
}
```

- Validation trade-offs
  - Too frequent to run in software (e.g. trap into OS)
  - Ideal validation policy – untaint operand only when user validates
  - Too strict ⇒ false positives ⇒ performance loss
  - Too lenient ⇒ false negatives ⇒ security flaw

Input Validation Flaws: DIFT

- Validation on pointer arithmetic (scaled addition)
  - Add untainted base + a tainted index ⇒ untainted pointer

- Problems
  - Several ISAs don't have scaled addition instructions
    - Untainting on regular adds ⇒ false positives
  - Several compilers use scaled addition instructions for integers
    - Incorrect validation ⇒ false negatives
  - Tainted indices can produce arbitrary values
    - Without bounds checking ⇒ false negatives
    - No correlation between pointer arithmetic and bounds checks!

Input Validation: PTD

- Validation on comparisons
  - Tainted pointer compared to untainted value ⇒ untainted value
- Strengths
  - Covers most bounds checks
  - No dependence on exotic instructions in ISA, compiler
  - Untaint directly associated with bounds check
- Problems
  - Bounds checks may have multiple forms
    - Integer truncation, masking bits, …
    - Missing validation ⇒ false positives
  - Bugs in comparisons
    - Signedness, translation tables…
    - Incorrect validation ⇒ false negatives

Format String Bugs

- Exploit format string directives for printf(), sprintf(), …
  - Untrusted input passed as printf format string
  - printf(“%x”) reads arguments from stack
  - %n directive writes number of chars output to its argument
  - %d will pad output with spaces if given width specifier

- Classical attack examples
  - E.g., printf(“%12345d%n”,17,&x) writes 12345 to x
  - E.g., printf(“\x41\x42\x43\x44%12345d%0d%0%d%0d%n”)
    - Write address 0x44434241 if format string begins at 5th word from top of stack

- Prevention by tainting architectures
  - Can write arbitrary value but will be tainted because derived from fmt string
  - Cannot place target address in fmt string because fmt string tainted

New Format Sting Attack

- Write an arbitrary untainted value to an arbitrary address
  - Avoid the use of a constant field width
  - Create one from existing untainted values
- Ingredients
  - "*" specifier to get field width from the stack
  - printf positional parameters to use arguments multiple times
  - Two pointer pairs that reference adjacent halfwords
    - Created with environment variables if local
- Tainting architectures will not catch this
  - The new arbitrary value is not tainted!

Information Leakage Prevention

- Problem: ensure confidentiality of data
  - Prevent untrusted applications from leaking information
  - E.g., cannot send private data to network
- RIFLE overview [Vachharajani'04]
  - Each information type has a security label
  - Labels propagate through instruction execution
  - Forbid sensitive labels from being sent to untrusted output channels

Implicit Information Flow

- Must track all information flow to prevent leaks
  - Implicit information flow occurs through control dependences

```
if (a)
   b = 0;
else
   b = 1;
```

- RIFLE implicit information flow tracking
  - Security registers track labels of any operand used as branch condition
  - Instructions for update of security registers inserted before each branch
  - Add label of branch operand to security register

Implicit Information Flow: Flaws

- No memory corruption protection
  - Attacker can corrupt code pointers to jump over security register update and just execute branch!
  - If security register has not been updated, implicit information flow occurs undetected by RIFLE
- Implications
  - Untrusted application may leak arbitrary amounts of sensitive information by corrupting own code pointers
  - If attacking nonmalicious application, only succeed if security register does not already contain label of branch operand

Conclusion

- Hardware support can greatly improve system security
  - High precision analysis with low overhead
  - Compatibility with existing software

- Need more flexible hardware security solutions
  - Flexibility in setting check/propagation policies
    - No one policy will catch all our bugs
  - Concurrent use of multiple types of checks
    - Different rules, hardware/software checks, …
  - Beyond memory corruption and information leakage
    - SQL injection, Cross Site Scripting, Directory Traversal, …