Using DISE to Protect Return Addresses from Attack

Marc L. Corliss, E Christopher Lewis, Amir Roth
University of Pennsylvania
Overview

- Prevent stack-smashing attacks
- Old approach, new implementation
  - Dynamic Instruction Stream Editing (DISE)
Stack-Smashing Attacks

\[ \text{strcpy}(A, \text{input\_string}); \]
Stack-Smashing is a Problem

- Some famous stack-smashing attacks
  - Internet worm, 1987
  - NCSA HP-UX 1.3 web server breach, 1994
  - Slammer worm, 2003
  - Blaster worm, 2003
Solutions

- Prevent buffer-overflow in general
  - Safe languages (e.g. Java, ML)
  - Range checks (e.g., libSafe)

- Prevent return address corruption
  - Canary: padding word (e.g., StackGuard)
  - Virtual memory: write protect (e.g., StackGuard)
  - **Shadow stack**
Talk Outline

- Introduction
- DISE background
- DISE return address protection
- Evaluation
- Conclusion
DISE

- Programmable instruction macro-expander
  - Like hardware SED (DISE = dynamic instruction SED)
  - Example: memory fault isolation (MFI)

```
srli r9, 4, r1
cmp r1, r2, r1
bne r1, Error
store r4, 8(r9)
```
DISE Productions

- Production: static rewrite rule

  \[
  \text{T.OPCLASS==store} \\
  \Rightarrow \quad \text{srl} \ T.RS,4,dr0 \\
  \text{cmp} \ dr0,dr1,dr0 \\
  \text{bne} \ dr0,\text{Error} \\
  \text{T.INST}
  \]

- Expansion: dynamic instantiation of production

  \[
  \text{store} \ r4,8(r9) \\
  \text{srl} \ r9,4,dr0 \\
  \text{cmp} \ dr0,dr1,dr0 \\
  \text{bne} \ dr0,\text{Error} \\
  \text{store} \ r4,8(r9)
  \]
Talk Outline

- Introduction
- DISE background
- DISE return address protection
- Evaluation
- Conclusion
Shadow Stack Approach

main:

RA

foo:

RA

strcpy:

RA

RA

RA

RA

RA

RA

RA

RA

RA==RA?
DISE Implementation

- At call instructions
  - Push return address on shadow stack
- At return instructions
  - Pop address from shadow stack
  - Verify shadow address equals return address
Protection Productions

T.OPCLASS==call

=> add T.PC,4,dr0  # compute RA
    add dssp,8,dssp  # push it on...
    store dr0,-8(dssp)  # ... shadow stack
    T.INST  # perform call

T.OPCLASS==return

=> load dr0,-8(dssp)  # pop RA from...
    add dssp,-8,dssp  # ... shadow stack
    cmpne T.RS,dr0,dr0  # cmp to actual RA
    ccall dr0,Error  # diff? then error
    T.INST  # perform return

Shadow stack:

RA1
RA2
RA3

dssp →
Implementation Issues

- Discussed in this talk
  - Handling non-local returns
  - Protecting shadow stack itself
- Discussed in the paper
  - Expanding DISE memory
  - Protecting DISE productions
Non-Local Returns

- Problem: Exceptions, setjmp/longjmp
  - Cut the stack
- Previous solution: pop shadow stack until match
  - Attacker can simulate longjmp
- Ours: store stack pointer (SP) on shadow stack
  + SP is unique in caller stack, not-smashable
  - Pop shadow stack until RA & SP match

New shadow stack:

```
    dssp
   ↓
RA1  SP1
RA2  SP2
RA3  SP3
```
New Productions

T.OPCLASS==call
=> add T.PC,4,dr0  # compute RA
    add dssp,16,dssp  # push it on...
    store dr0,-8(dssp)  # ... shadow stack
    store sp,-16(dssp)  # ... along w/ stack ptr
    cmp dssp,darp,dr0  # stack full?
    ccall dr0,expand  # yes, expand stack
    T.INST  # perform call

T.OPCLASS==return
=> load dr0,-8(dssp)  # pop RA from...
    add dssp,-16,dssp  # ... shadow stack
    cmpne T.RS,dr0,dr0  # cmp to actual RA
    ccall dr0,AddrChk  # diff? figure out why
    T.INST  # perform return
Protecting the Shadow Stack

- What if attack corrupts shadow stack?
- One approach: encrypt shadow stack (XOR)

| T.OPCLASS==call | => | add T.PC,4,dr0 | # compute RA |
| | | xor dr0,dxr,dr0 | # encrypt address |
| | | add dssp,16,dssp | # push it on... |
| | | store dr0,-8(dssp) | # ... shadow stack |
| | | store sp,-16(dssp) | # ... along w/ stack ptr |
| | | T.INST | # perform call |

| T.OPCLASS==return | => | load dr0,-8(dssp) | # pop RA from... |
| | | add dssp,-16,dssp | # ... shadow stack |
| | | xor dr0,dxr,dr0 | # decrypt address |
| | | cmpne T.RS,dr0,dr0 | # comp. to actual RA |
| | | ccall dr0,AddrChk | # diff? See what’s up |
| | | T.INST | # perform return |
Another Approach

- Prevent writes to shadow stack (e.g. MFI)
- Call/return productions don’t change
- New productions for stores

\[
\text{T.OPCLASS==store} \\
\Rightarrow \text{srl} \text{ T.RS,4,dr0} \\
\text{cmp dr0,dr1,dr0} \\
\text{beq dr0,Error} \\
\text{T.INST}
\]
Virtues of Using DISE

- Versus dedicated hardware
  + General-purpose: compression, debug., mfi, etc.
  + Flexible: new attack, new productions

- Versus binary rewriter
  + Transparent: protect all code inc. DLLs
  + Declarative interface: security by simplicity
  + Efficient: in a moment…
Talk Outline

- Introduction
- DISE background
- DISE return address protection
- Evaluation
- Conclusion
Evaluation

- Correctness
  - False negatives?
  - False positives?

- Performance
  - What’s the overhead?
  - How does it compare to alternatives?
Correctness

- Attacking inputs detected
  - overflow1 (micro-benchmark)
  - sendmail-8.7.5
  - gzip-1.2.4
- No false positives with non-attacking inputs
Performance Methodology

- SimpleScalar Alpha
- Benchmarks
  - SPEC Int 2000, MiBench, CommBench
- Binary rewriter (for comparison)
  - Statically insert DISE instrumentation code
  ± No optimization, but not a lot of opportunity either
Performance Overhead

- DISE overhead reasonable
  - XOR less than 10%, MFI less than 35%
- DISE outperforms BR
  - BR has additional I$ misses
Conclusion

- DISE return address protection effective
  - No false negatives, no false positives
- Advantages over binary rewriter
  - Efficient, transparent, declarative interface
- Advantages over custom hardware
  - General-purpose, flexible

- Future work
  - Other security-related applications using DISE