Neutralizing Manipulation of Critical Data by Enforcing Data-Instruction Dependency

Chandra Sharma
Nathan Miller
George Amariucai

Introduction

- A program encapsulates several critical data
  - Influential in determining the control flow of a program
- Manipulation of the critical data:
  - Allows access to locked software features
  - May even allow full system access
- Imperative to protect such data from illicit modification

Background

- One class of critical data that is often subject to malicious manipulation: the *return address* of a function
  - Part of the activation record saved on the stack
  - Determines program control flow
  - Primary subject of stack-smashing attacks
- Manipulation of the return address allows diversion of program control

Existing Measures

- Mostly based on either randomization of an address/pointer parameter or secrecy of some random value/key
  - ASLR, ASLP, ILR
    - Based on randomization of addresses
    - Vulnerable to memory disclosure
  - StackGuard
    - Relies on the secrecy of canary
    - Vulnerable to buffer-overread, canary bypass
  - Instruction Set Randomization
    - Randomizes the entire instruction set
    - High overhead
    - Vulnerable to chosen-key attacks, code-reuse attacks
  - Return Address Defender
    - Uses parallel stacks
    - Robust but less versatile

Data-Instruction Dependency

- Does not rely on randomization
- Does not rely on a secret value/key
- Centered around the notion of critical instructions
  - Instructions that determine continuation/ termination of a program
- Critical instructions are used as a trap against illicit modification of critical data

RAID

- Set up a dependency between the return address and some sequence of instructions
- Goal:
  - Execution of the instructions succeeds if the return address is intact
  - Execution fails otherwise

Example

```c
int add(int x, int y)
{
  int result = x+y;
  return result;
}

int main()
{
  int sum = add(10,20);
  return 0;
}
```

```asm
add:
  push ebp
  mov ebp, esp
  sub esp, 4
  mov edx, DWORD PTR 8[ebp]
  mov eax, DWORD PTR 12[ebp]
  add eax, edx
  mov DWORD PTR -4[ebp], eax
  mov eax, DWORD PTR -4[ebp]
  leave
  ret
```

Implementation

- Encode a sequence of (critical) instructions at the start of a function with the return address
- Decode right before the function returns and execute the decoded sequence
- If the return address is tampered with between the encoding and decoding steps, the execution fails resulting in a program crash
- Successful execution of critical instructions preserve program semantics

Code Stack

- Allocate a separate stack space, a *code stack*, for critical instructions
  - Each function is allocated a frame in the code stack
- Encoding and decoding operations are performed in the code stack
- Critical instructions are copied to the code stack at the start of the function’s (modified) prologue

Example

Original:

```
add:
  push ebp
  mov ebp, esp
  sub esp, 4
  mov edx, DWORD PTR 8[ebp]
  mov eax, DWORD PTR 12[ebp]
  add eax, edx
  mov DWORD PTR -4[ebp], eax
  mov eax, DWORD PTR -4[ebp]
  leave
  ret
```

Modified:

```
add:
  * Encode the ret instruction
  * Copy to the code stack
  push ebp
  mov ebp, esp
  sub esp, 4
  mov edx, DWORD PTR 8[ebp]
  mov eax, DWORD PTR 12[ebp]
  add eax, edx
  mov DWORD PTR -4[ebp], eax
  mov eax, DWORD PTR -4[ebp]
  leave
  * Decode the ret instruction
  * Jump to the code stack
```

Code Stack Illustration

F4	F4	F4	F4
F4	F4	F4	...

Fig. 1: Code stack right before the execution of the modified prologue

E7	F4	F4	F4
F4	F4	F4	...

Fig. 2: Code stack right after the execution of the modified prologue

mov bl, 0xF4

B3	F4	F4	F4
F4	F4	F4	...

Fig. 3: Code stack resulting from incorrectly decoded *ret* instruction

ret

C3	F4	F4	F4
F4	F4	F4	...

Fig. 4: Code stack resulting from correctly decoded *ret* instruction

03 F4 F4 ...	add esi, esp hlt ...	83 F4 F4 F4 ...	xor esp, 0xFFFFFFF4 hlt ...
13 F4 F4 ...	adc esi, esp hlt ...	93 F4 ...	xchg ebx, eax hlt ...
23 F4 F4 ...	and esi, esp hlt ...	A3 F4 F4 F4 F4 F4 ...	mov ds:0xF4F4F4F4, eax hlt ...
33 F4 F4 ...	xor esi, esp hlt ...	B3 F4 F4 ...	mov bl, 0xF4 hlt ...
43 F4 ...	inc ebx hlt ...	C3 ...	ret ...
53 F4 ...	push ebx hlt ...	D3 F4 ...	invalid opcode hlt ...
63 F4 F4 ...	arpl sp, si hlt ...	E3 F4 F4 ...	jecxz -10 hlt ...
73 F4 F4 ...	jae -10 hlt ...	F3 F4 ...	repz hlt ...

Fig. 5: A list of all possibilities when the high nibble of the *ret* instruction is decoded

leave

- Precedes the *ret* instruction
- Releases the stack frame just before the function returns
- Positions the *esp* register to the saved return address

C0 C3 F4 F4 ...	rol bl, 0xF4 hlt ...	C8 C3 F4 F4 F4 ...	enter 0xF4C3, 0xF4 hlt ...
C1 C3 F4 F4 ...	rol ebx, 0xF4 hlt ...	C9 C3 ...	leave ret ...
C2 C3 F4 ...	ret 0xF4C3 ...	CA C3 F4 ...	retf 0xF4C3 ...
C3 ...	ret ...	CB ...	retf ...
C4 ...	invalid opcode ...	CC C3 ...	int 3 ret ...
C5 ...	invalid opcode ...	CD C3 F4 ...	int 0x03 hlt ...
C6 C3 F4 F4 ...	mov bl, 0xF4 hlt ...	CE C3 ...	into ret ...
C7 C3 F4 F4 F4 F4 F4 ...	mov ebx, 0xF4F4F4F4 hlt ...	CF ...	iret ...

Fig. 9: A list of some possibilities when the low nibble of the *leave* instruction is decoded

Fabricating Critical Instructions

- The *leave* and *ret* instructions constitute the epilogue of a function
- More instructions are needed for a complete dependency
- Introduce new instructions that do not break the semantics of the program

An Example

Original function epilogue:

```
add:
  …
  leave
  ret
```

Modified function epilogue:

```
add:
  …
  sub esp, someOffset
  push ebp
  mov ebp, esp
  mov esp, 0
  inc ebp
  dec ebp
  leave
  leave
  ret
```

40 C9 C9 ...	inc eax leave leave ...	48 C9 C9 ...	dec eax leave leave ...	44 C9 C9 ...	inc esp leave leave ...	4C C9 C9 ...	dec esp leave leave ...
41 C9 C9 ...	inc ecx leave leave ...	49 C9 C9 ...	dec ecx leave leave ...	45 C9 C9 ...	inc ebp leave leave ...	4D C9 C9 ...	dec ebp leave leave ...
42 C9 C9 ...	inc edx leave leave ...	4A C9 C9 ...	dec edx leave leave ...	46 C9 C9 ...	inc esi leave leave ...	4E C9 C9 ...	dec esi leave leave ...
43 C9 C9 ...	inc ebx leave leave ...	4B C9 C9 ...	dec ebx leave leave ...	47 C9 C9 ...	inc edi leave leave ...	4F C9 C9 ...	dec edi leave leave ...

Fig. 10: A list of some possibilities when the low nibble of the *dec ebp* instruction is encoded/decoded

ucc_RAID

- Unoptimized prototype compiler implementing RAID
- Splits the program's stack into two partitions
  - Regular Stack
  - Code Stack
    - Located at offset 0x10000 from the regular stack
- Incurs negligible compile-time overhead
- Does incur notable run-time overhead
  - Can be significantly reduced with operating system support

Thank You

- Contact us at:
  - Chandra Sharma: ch1ndra@ksu.edu
  - Nathan Miller: nathan232@ksu.edu
  - George Amariucai: amariucai@ksu.edu