SmashClean: A Hardware level mitigation to stack smashing attacks in OpenRISC

Team : *SEAL*

Secured Embedded Architecture Laboratory
Indian Institute of Technology Kharagpur

Cyber Security Awareness Week 2016
Embedded Security Challenge
11th November, 2016

Outline

Introduction

- **Security threats to Embedded Systems**
  - Performance-efficient languages such as C and C++ are widely used for embedded applications.
  - Vulnerable to memory corruption due to lack of secure memory management.
- **Buffer Overflow**
  - Triggers malicious code execution by overwriting correct memory content.
  - Software level countermeasures can be easily bypassed.
  - Need hardware level countermeasures (e.g., hardware-based protection of the function return address).
  - Existing architectures target platform different from the OpenRISC ISA processor.

Objective

SmashClean

Design Hardware-Based Mitigation Technique of Memory Corruption and Ensuring Control Flow Integrity for the OpenRISC ISA Processor.

Exploitation Methods

- *The root cause of buffer overflow threat* : `memcpy()` does not impose any bound-checking during memory update.

Types of Exploitation

- Control Flow Modification.
  - Return Address Modification (`stack.c`).
  - Format String Vulnerability (`format.c`).
- Memory Corruption.
  - Data Pointer Modification (`priv.c`).
  - Function Pointer Modification (`ptr.c`).

Exploitation Methods

stack.c

```
int func(char* user, int len) {
    char buff[100];
    memcpy(buff, user, len); //Vulnerability
}
```

Control Flow Modification

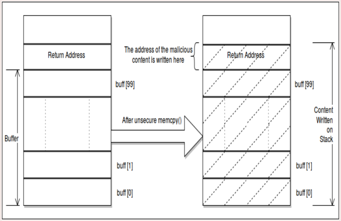

Figure: Return address modification by Buffer Overflow

priv.c

```
int func(char* user, int len) {
    int *ptr;
    int newdata = 0xaaaa;
    char buff[16];
    int olddata = 0xffff;
    ptr = &olddata;
    memcpy(buff, user, len); //Vulnerability
    *ptr = newdata;
}
```

Memory Corruption

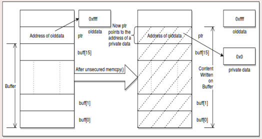

Figure: Data Pointer modification by Buffer Overflow

Protect Control Flow

- Implementation of a hardware stack which stores the function return address for each of the function.

Assembly Code of `stack.c`

```
vuln:
.LFB1:
    .cfi_startproc
        ⋮
    l.ori r1,r2,0 # deallocate frame
    l.lwz r2,-8(r1) # SI load
    l.lwz r9,-4(r1) # SI load
    l.jr r9 # return_internal
    l.nop # nop delay slot
    .cfi_endproc
```

Prevention using Hardware Stack

- Whenever it encounters a `l.jal` or `l.jalr` instruction, it pushes the next program counter value to the stack.
- Alternatively if it encounters `l.jr` instruction with register `r9` as parameter, it pops its top value and passes that as the return address.
- Custom instruction `l.cust1`, when enabled, ensures that the return address of the functions are read from the hardware stack.
- Custom instruction `l.cust2` disables the hardware stack.

Prevent Memory Corruption

- We introduced hardware enforced secure `memcpy()`.
- This protection prevents buffer overflow by hardware induced bound check and prevents any memory corruption due to buffer overflow.

Assembly Code of `priv.c`

```
vuln:
l.sw -40(r2),r3 # SI store
        ⋮
l.sw -36(r2),r3 # SI store
        ⋮
l.nop # nop delay slot
l.lwz r4,-44(r2) # SI load
l.addi r3,r2,-32 # addsi3
l.ori r5,r4,0 # move reg to reg
l.lwz r4,-40(r2) # SI load
l.jal memcpy # call_value_internal
l.nop # nop delay slot
```

Prevention Procedure

- The first instruction (`l.addi r3, r2, -32`) transfers the starting address of the buffer (`r2 - 32`) to `r3`. The address of the latest new variable in this case is `r2 - 16`. Subtracting this two will give us buffer size which in this case is 16.
- The next instruction `l.ori` transfers the function argument count to `r5` which denotes the number of memory locations to be updated by `memcpy()`.
- Now, we will check whether the instruction `l.ori r5, r4, 0` returns the count value greater than the buffer size or not.

IIT Kharagpur

Proposed Architecture

Proposed Hardware Stack

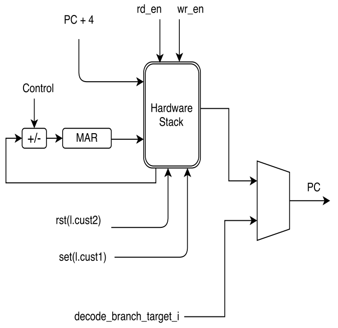

Figure: Hardware Stack

Secure `memcpy()` function

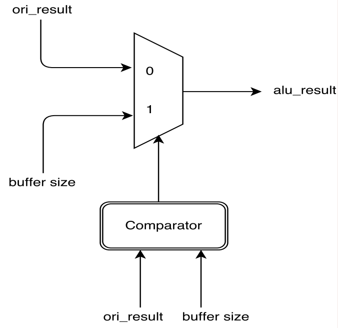

Figure: Secure `memcpy()`

New Instructions

- `l.cust3` : This instruction will be inserted by the compiler just before `memcpy()` function is declared in C code to protect buffer overflow. This instruction sets a specific flag inside the processor and observes the occurrence of `l.addi` and `l.ori` which are required for computation of buffer size. If the buffer size is less than the argument count a `smash_detect` flag is set and the value of the count argument is updated with the buffer size.
- `l.cust4` : This instruction resets the `smash_detect` flag.
- `l.cust5` : This instruction induces a lock on latest variable address location to preserve it from intermediate function calls. This can be alternatively achieved by maintaining a hardware stack for latest variable locations for each function call.
- `l.cust6` : This instruction removes the aforementioned lock.

Conclusion

- Prevented popular forms of memory corruption and buffer overflow attacks on OpenRISC architecture.
- Combined compiler and hardware modification.
- Introduced new instructions via hardware modification for compiler to detect and prevent memory corruption via buffer overflow.