# 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
$11^{th}$ 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

- Buffer overflow occurs when a program attempts to read or write beyond the end of a bounded array (also known as a buffer)

## Buffer Overflow

| Variable | A | | | | | | | | B | |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | null string | | | | | | | | 2016 | |
| Hex Value | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 07 | E0 |

strcpy(A, "sealiitkgp")

| Variable | A | | | | | | | | B | |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | 's' | 'e' | 'a' | 'l' | 'i' | 'i' | 't' | 'k' | 26480 | |
| Hex Value | 73 | 65 | 61 | 6C | 69 | 69 | 74 | 6B | 67 | 70 |

Figure: Example of Buffer overflow

# Exploitation Methods

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

## Types of Exploitation

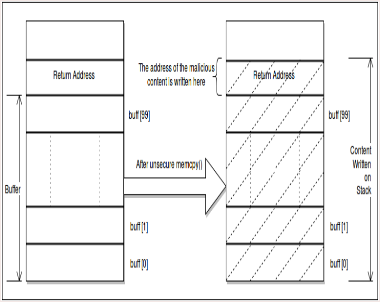
- Control Flow Modification.
  - Return Address Modification (`stack.c`).
  - Format String Vulnerability (`format.c`). (doesn't use `memcpy()`)
- Memory Corruption.
  - Data Pointer Modification (`priv.c`).
  - Function Pointer Modification (`ptr.c`).

# Exploitation Methods

## stack.c

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

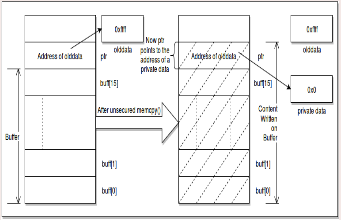
## Control Flow Modification




## 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



# Exploitation Methods

## format.c

```
int func(char* user) {
        .
        .
        .
    printf(user); //Vulnerability
        .
        .
        .
}
```

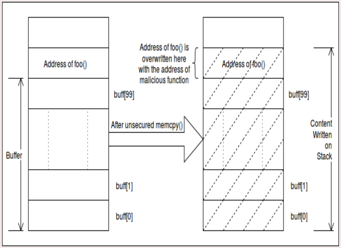
## Vulnerabilities of printf

- printf is a *varargs* function.
  int printf(const char *format, ...);
- printf("%p") will print out data from stack memory.
- Reveals information about the state of program's memory to an attacker.

## ptr.c

```
int func(char* user, int len) {
    void (*fptr)(char *);
    char buff[100];
    fptr = &foo; //Address of intended function
    memcpy(buff, user, len); //Vulnerability
    fptr(user);
}
```

## Memory Corruption



# Memory Allocation

## Memory Allocation inside OpenRISC

| Position | Contents | Frame |
|---|---|---|
| FP+4N<br>...<br>FP+0 | Parameter N<br>...<br>First stack parameter | Previous |
| FP-4 | Return address | Current |
| FP-8 | Previous FP Value | Current |
| FP-12<br>...<br>SP+0 | Function variables<br>...<br>Subfunction call parameters | Current |
| SP-4<br>SP-128 | For use by leaf functions w/o function prologue/epilogue parameters | Future |
| SP-132<br>SP-2536 | For use by exception handlers | Future |

# Protection using Hardware Stack

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

## Prevention Procedure

- 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.

# Protection using Hardware Stack

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

**Compiler Modified Code of format.c**

```
int main(int argc, char **argv) {
        .
        .
        .
    asm volatile("l.cust7");
    vuln(fstr, count);
    asm volatile("l.cust2");
}
void vuln(char* s, int offset) {
        .
        .
        .
    asm volatile("l.cust8");
    printf("Parsing starts at address %p", ptr); //First printf()
        .
        .
        .
    printf(s); //vulnerable (Last printf())
    asm volatile("l.cust1");
}
```

**Custom Instructions Used**

- l.cust7 ensures that the return address of the functions are read from the hardware stack.
- l.cust8 freezes the hardware stack.
- l.cust1 unfreezes the hardware stack.
- l.cust2 disables the hardware stack.

# Proposed Architecture

## Proposed Hardware Stack

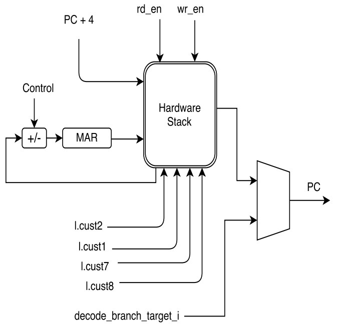



Figure: Hardware Stack

## Secure memcpy() function

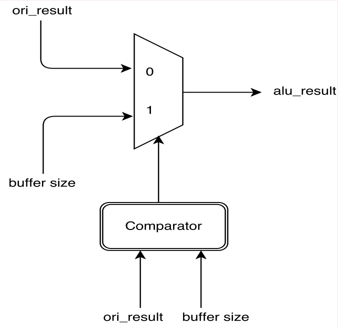



Figure: Secure memcpy()

# Protection using secure memcpy()

**Assembly Code of stack.c**

```
vuln:
l.sw -28(r2),r3 # SI store
l.sw -32(r2),r4 # SI store
        .
        .
        .
l.lwz r4,-32(r2) # SI load
l.addi r3,r2,-24 # addsi3
l.ori r5,r4,0 # move reg to reg
l.lwz r4,-28(r2) # SI load
l.jal memcpy # call_value_internal
l.nop # nop delay slot
```

**Assembly Code of ptr.c**

```
vuln:
l.sw -32(r2),r3 # SI store
l.sw -36(r2),r4 # SI store
        .
        .
        .
l.lwz r4,-36(r2) # SI load
l.addi r3,r2,-28 # addsi3
l.ori r5,r4,0 # move reg to reg
l.lwz r4,-32(r2) # SI load
l.jal memcpy # call_value_internal
l.nop # nop delay slot
```

**Assembly Code of priv.c**

```
vuln:
l.sw -40(r2),r3 # SI store
l.sw -44(r2),r4 # SI store
        .
        .
        .
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
```

# Protection using secure `memcpy()`

- 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.

## 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.

# Protection using secure memcpy()

## Compiler modified code of stack.c, ptr.c & priv.c

```
void vuln(char* s, int offset) {
        .
        .
        .
    asm volatile("l.cust5");
    printf("vuln() has received %d bytes", count);
    asm volatile("l.cust6");
    asm volatile("l.cust3");
    memcpy(buff, s, count);
    asm volatile("l.cust4");
        .
        .
        .
}
```

## Custom Instructions Used

- l.cust3 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 resets the smash_detect flag.
- l.cust5 induces a lock on latest variable address location to preserve it from intermediate function calls.
- l.cust6 removes the aforementioned lock.

# Proposed Architecture

## Proposed Hardware Stack

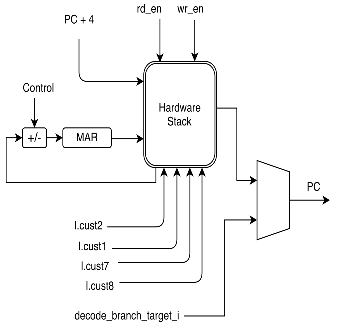



Figure: Hardware Stack

## Secure memcpy() function

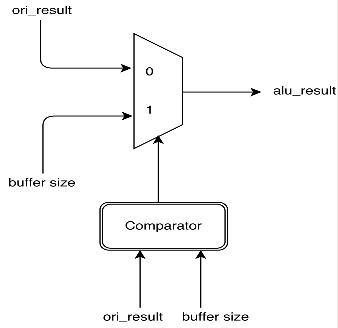



Figure: Secure memcpy()

# 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.

**Thank You**