COMP 737011 - Memory Safety and Programming Language Design

Lecture 4: Memory Exhaustion

Xu, Hui

xuh@fudan.edu.cn

Outline

- 1. Stack Overflow
- 2. Heap Exhaustion
- 3. Exception Handling
- 4. Stack Unwinding

1. Stack Overflow

Warm Up

- Can you find a list to overflow the stack?

```
struct List{
    int val;
    struct List* next;
};
```

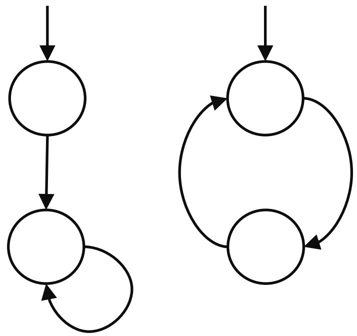
```
void process(struct List* l, int cnt){
    printf("%d\n", cnt);
    if(l->next != NULL)
        process(l->next, ++cnt);
}
```

- Sample solution

```
void main(void){
    sethandler(handler);
    struct List* list = malloc(sizeof(struct List));
    list->val = 1;
    list->next = list;
    process(list, 0);
}
```

Stack Size is Limited

- Default stack size: 8MB for each thread in Linux
  - You may check the setting with the ulimit command
- Reaching the limit would cause stack overflow
- Why not use a large stack?
  - Mainly used to save the contexts of function calls
  - Developers should not place large data on stack
- Vulnerable code: recursive function calls



```
#: ulimit -a
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
stack size              (kbytes, -s) 8192
max user processes              (-u) 30687
```

You May Change The Stack Limit

- System users: ulimit command

```
aisr@aisr:~$ ulimit -s unlimited
aisr@aisr:~$ ulimit -a
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
stack size              (kbytes, -s) unlimited
```

Set the stack size to unlimited

- Developers: use the setrlimit() function

```
struct rlimit r;
int result;
result = getrlimit(RLIMIT_STACK, &r);
fprintf(stderr, "stack result = %d\n", r.rlim_cur);
r.rlim_cur = 64 * 1024L *1024L;
result = setrlimit(RLIMIT_STACK, &r);
result = getrlimit(RLIMIT_STACK, &r);
fprintf(stderr, "stack result = %d\n", r.rlim_cur);
```

Set the stack size to 64 MB

How to Handle Stack Overflow?

- The OS usually kills the process directly. Why?
- We can register a handler for the SIGSEGV signal
- Executing the event handler needs an extra stack
  - need to register another stack with enough space
- You will learn this in your in-class practice

2. Heap Exhaustion

Overcommit

- A lazy mode memory allocation mechanism
  - malloc() successful does not mean the physical memory is allocated
  - The physical memory is allocated when being accessed
- Linux has three options
  - 1: always overcommit, never check
  - 2: always check, never overcommit
  - 0: heuristic overcommit (this is the default)

```
#: sudo sysctl -w vm.overcommit_memory=2
```

Overcommit: Example

- Try the following program with different settings

```
#define LARGE_SIZE 1024L*1024L*1024L*256L
void main(void){
    char* p = malloc (LARGE_SIZE);
    if(p == 0) {
        printf("malloc failed\n");
    } else {
        memset (p, 1, LARGE_SIZE);
    }
}
```

256 GB

allocation failure

allocation successful
access the memory

```
#: sudo sysctl -w vm.overcommit_memory=1
#:~/4-memoxhaustion$ ./a.out
Killed
```

killed by the OS

```
#: sudo sysctl -w vm.overcommit_memory=2
#:~/4-memoxhaustion$ ./a.out
malloc failed
```

allocation failure

How to Handle Heap Exhaustion?

- Always check: based on the return value of malloc()
  - returns 0 if fails
- Overcommit: could be killed by the OS
  - register a handler for the SIGKILL signal?
- To Small to Fail & OOM Killer
  - If the required space is small (< 8 pages), malloc() should never fail when overcommit is enabled
  - If no enough memory, a process would be killed by the OOM killer
    - based on badness of each process
    - calculated based on the vmsize and uptime of each process

The "too small to fail" memory-allocation rule, https://lwn.net/Articles/627419/

To Small to Fail: Example

```
#define SMALL_SIZE 1024L
void exhaustheap() {
    for(long i=0; i < INT64_MAX; i++) {
        char* p = malloc (SMALL_SIZE);
        if(p == 0){
            printf("the %ldth malloc failed\n", i);
            break;
        } else {
            printf("access the %ldth memory chunk,...", i);
            memset (p, 0, sizeof (SMALL_SIZE));
            printf(", done\n", i);
        }
    }
}
```

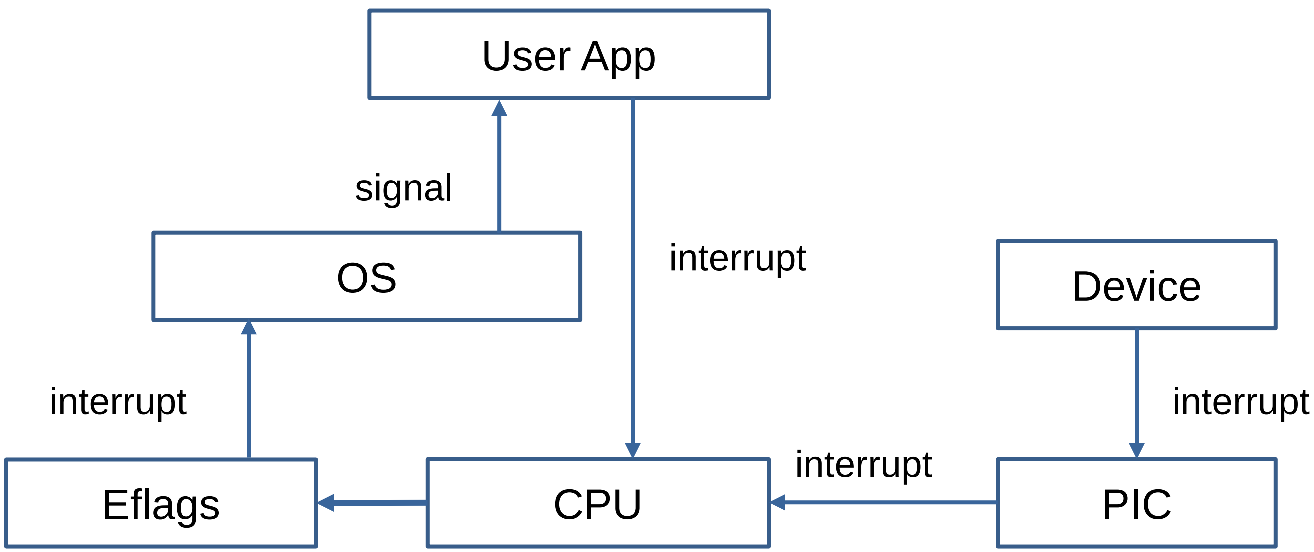
```
#: sudo sysctl -w vm.overcommit_memory=2
#:~/4-memoxhaustion$ ./a.out
...
access the 2705176th memory chunk,..., done
the 2705177th malloc failed
```

```
#: sudo sysctl -w vm.overcommit_memory=1
#:~/4-memoxhaustion$ ./a.out
...
access the 9013022th memory chunk,..., done
Killed
```

3. Exception Handling

Exceptions based on Origin

- CPU: interrupt
- OS: signal
- Application: user-defined exceptions



CPU Interrupt

- Page fault, divided by zero, etc
- Jump to the target exception handling address based on an interrupt vector, e.g., for X86
  - 0x00 Division by zero
  - 0x01 Single-step interrupt (see trap flag)
  - 0x03 Breakpoint (INT 3)
  - 0x04 Overflow
  - 0x06 Invalid Opcode
  - 0x0B Segment not present
  - 0x0C Stack Segment Fault
  - 0x0D General Protection Fault
  - 0x0E Page Fault
  - 0x10 x87 Floating Point Exception

OS Signal

- Kernel sends to other processes (IPC)
- POSIX signals
  - SIGFPE: floating-point error, overflow, underflow...
  - SIGSEGV: segmentation fault, invalid address...
  - SIGBUS: bus error, memory alignment issue
  - SIGILL: illegal instruction
  - SIGABRT: abort
  - SIGKILL:
  - …

Register the OS Signal

- Register the OS signal with signal or sigaction

```
void sethandler(void (*handler)(int,siginfo_t *,void *)){
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigaction(SIGFPE, &sa, NULL);
}

void handler(int signo, siginfo_t *info, void *extra){
    printf("SIGFPE received!!!\n");
    exit(-1);
}

int main(void){
    sethandler(handler);
    int a = 0;
    int x = 100/a;
}
```

https://man7.org/linux/man-pages/man2/sigaction.2.html

Exception Handling Issue

- Where should the process continue?
  - find a landing pad
- How to set the required execution context?
  - restore callee-saved registers: rbp、rsp、rbx、r12-r15
- Release acquired resources
  - e.g, heap, file discriptor

setjmp/longjmp

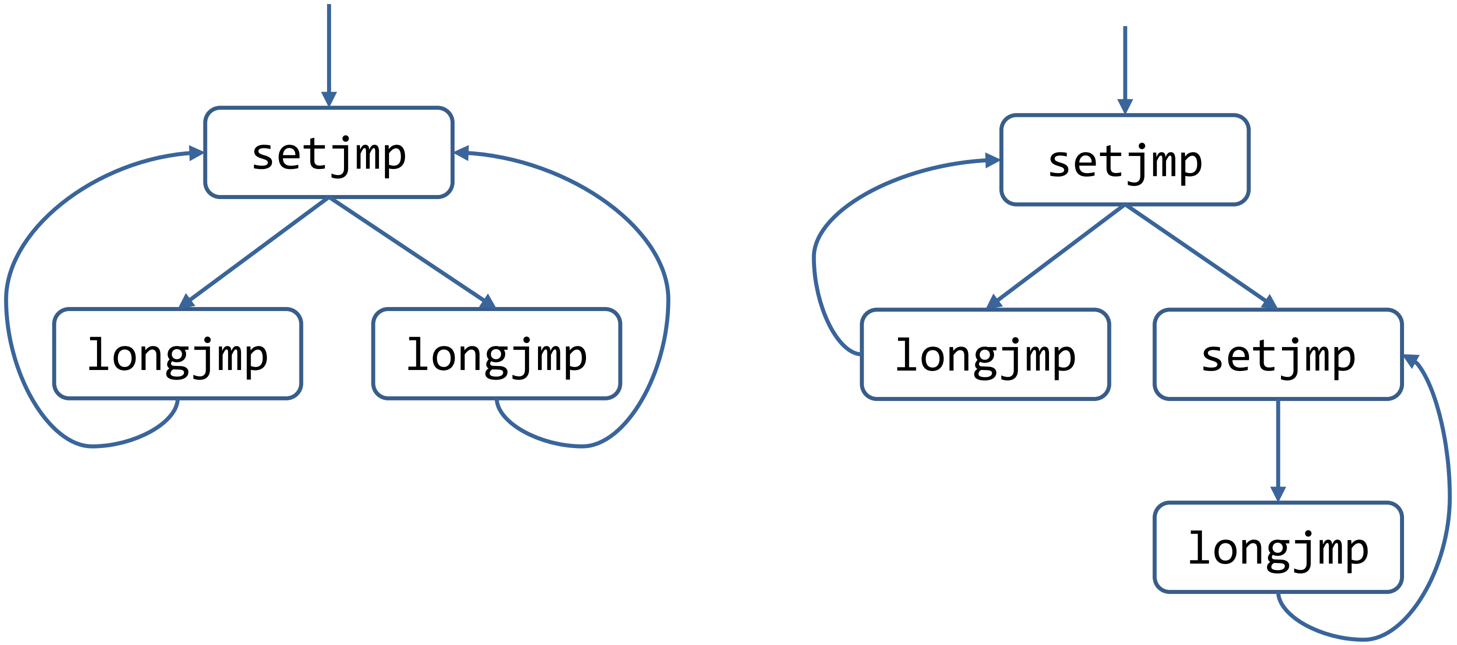
- setjmp(env)：
  - backup registers and sets a recover point
  - return 0 if called directly, otherwise return a value if called by longjmp()
- longjmp(env,value)：
  - jump to a target address determined by value
  - restore all callee-saved registers: rbp、rsp、rbx、r12-r15

```
static jmp_buf buf;
void handler(int signo, siginfo_t *info, void *extra){
    printf("SIGFPE received!!!\n");
    longjmp(buf,1);
}

int main(void){
    sethandler(handler);
    int a = 0;
    if (!setjmp(buf))
        int x = 100/a;
    else
        printf("Contine execution after a longjmp.\n");
}
```

Discussion

- How to support multiple setjmps/longjmps?

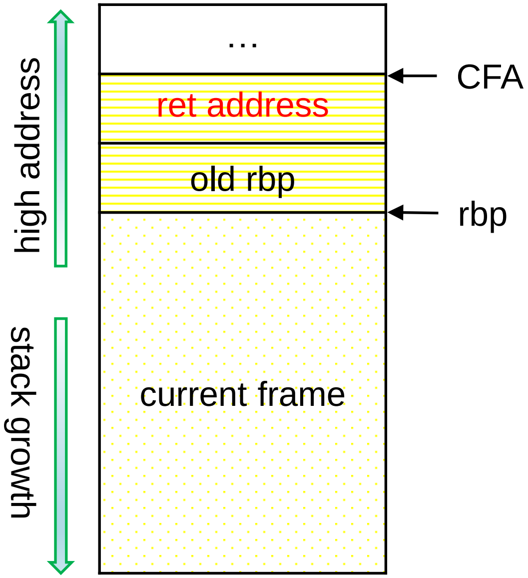


4、 Stack Unwinding

Problem

- Callee-saved registers should be restored
- Setjmp/longjmp is inconvienent or inefficient if widely used
- Can we have a better solution?

```
0x401130: push   %rbp
0x401131: mov    %rsp,%rbp
0x401134: sub    $0x10,%rsp
0x401138: mov    %edi,-0x8(%rbp)
0x40113b: cmpl   $0x0,-0x8(%rbp)
0x40113f: jne    0x401151
0x401145: movl   $0x1,-0x4(%rbp)
0x40114c: jmpq   0x40116d
0x401151: mov    -0x8(%rbp),%eax
0x401154: mov    -0x8(%rbp),%ecx
0x401157: sub    $0x1,%ecx
0x40115a: mov    %ecx,%edi
0x40115c: mov    %eax,-0xc(%rbp)
0x40115f: callq  0x401130
0x401164: mov    -0xc(%rbp),%ecx
0x401167: imul   %eax,%ecx
0x40116a: mov    %ecx,-0x4(%rbp)
0x40116d: mov    -0x4(%rbp),%eax
0x401170: add    $0x10,%rsp
0x401174: pop    %rbp
0x401175: retq
```



DWARF

- Calculate the information required for recovering from each instruction during compilation
- Such data format (DWARF) and mechanism is defined in the standard of ABI
- The program unwinds the call stack iteratively
- Different from the dynamic solution with setjmp.
  - more convenient, throw/try/catch is based on DWARF
  - more efficient

How Does DWARF Work?

- To recover the context of the caller, we should know whether callee-saved registers have been changed
- Such callee-saved registers should be saved on the stack
- Record the address of each callee-saved register

Example

- Calculate the canonical frame address or CFA
  - Find all instructions related to stack expansion/reduction
- Record the address of callee-saved registers related to CFA

```
push    %rbp
mov     %rsp,%rbp
sub     $0x10,%rsp
mov     %edi,-0x8(%rbp)
cmpl    $0x0,-0x8(%rbp)
jne     0x401151
movl    $0x1,-0x4(%rbp)
jmpq    0x40116d
mov     -0x8(%rbp),%eax
mov     -0x8(%rbp),%ecx
sub     $0x1,%ecx
mov     %ecx,%edi
mov     %eax,-0xc(%rbp)
callq   0x401130
mov     -0xc(%rbp),%ecx
imul    %eax,%ecx
mov     %ecx,-0x4(%rbp)
mov     -0x4(%rbp),%eax
add     $0x10,%rsp
pop     %rbp
retq
```

push %rbp → return address = CFA-8
CFA = cur rsp + 16, old rbp = CFA – 16,

sub $0x10,%rsp → CFA = cur rsp + 32

add $0x10,%rsp → CFA = cur rsp + 16;

pop %rbp → CFA = cur rsp – 8, old rbp is already restored

Check DWARF Data with pyreadelf

- Saved in the eh_frame section of ELF files

```
python3 pyelftools-master/scripts/readelf.py --debug-dump frames-interp /bin/cat
```

```
2690: endbr64
2694: push   %r15
2696: mov    %rsi,%rax
2699: push   %r14
269b: push   %r13
269d: push   %r12
269f: push   %rbp
26a0: push   %rbx
26a1: lea    0x4f94(%rip),%rbx
26a8: sub    $0x148,%rsp
26af: mov    %edi,0x2c(%rsp)
26b3: mov    (%rax),%rdi
…
27e7: sub    $0x8,%rsp
…
27fb: pushq  $0x0
…
2e96: pop    %rbx
2e97: pop    %rbp
2e98: pop    %r12
2e9a: pop    %r13
2e9c: pop    %r14
2e9e: pop    %r15
2ea0: retq
```

```
   LOC    CFA      rbx   rbp   r12   r13   r14   r15   ra
00002690 rsp+8     u     u     u     u     u     u     c-8
00002696 rsp+16    u     u     u     u     u     c-16  c-8
0000269b rsp+24    u     u     u     u     c-24  c-16  c-8
0000269d rsp+32    u     u     u     c-32  c-24  c-16  c-8
0000269f rsp+40    u     u     c-40  c-32  c-24  c-16  c-8
000026a0 rsp+48    u     c-48  c-40  c-32  c-24  c-16  c-8
000026a1 rsp+56    c-56  c-48  c-40  c-32  c-24  c-16  c-8
000026af rsp+384   c-56  c-48  c-40  c-32  c-24  c-16  c-8
000027eb rsp+392   c-56  c-48  c-40  c-32  c-24  c-16  c-8
000027fd rsp+400   c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002825 rsp+384   c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e96 rsp+56    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e97 rsp+48    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e98 rsp+40    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9a rsp+32    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9c rsp+24    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9e rsp+16    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002ea0 rsp+8     c-56  c-48  c-40  c-32  c-24  c-16  c-8
```

Usage of DWARF

- Debuging: developers can obtain the call stack with backtrace()
- Exception handling: require further information to determine the landing pad or language specific information (personality routine)
  - C++ try-throw-catch
  - Rust stack unwinding

In-Class Practice

- Handle the stack overflow issues of the following code
- You task is to implement the sethandler and handler functions
  - Useful APIs: setjmp/longjmp, sigaction, sigaltstack
  - ref: https://man7.org/linux/man-pages/man2/sigaltstack.2.html

```
struct List{
    int val;
    struct List* next;
};
void process(struct List* list, int cnt){
    if(list->next != NULL)
        process(list->next, ++cnt);
}
void sethandler(void (*handler)(int,siginfo_t *,void *))
void handler(int signo, siginfo_t *info, void *extra);
void main(void){
    sethandler(handler);
    struct List* list = malloc(sizeof(struct List));
    list->val = 1;
    list->next = list;
    if (setjmp(buf) == 0)
        process(list, 0);
    else
        printf("Continue after segmentation fault\n");
}
```