COMP 737011 - Memory Safety and Programming Language Design

Lecture 4: Memory Exhaustion and Exception Handling

徐 辉
xuh@fudan.edu.cn

Background

- Processes of memory exhaustion will be killed by the OS kernel.
  - Stack Overflow
  - Heap Exhaustion
- How can we prevent such unexpected termination?

Outline

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

1. Stack Overflow

Warm Up

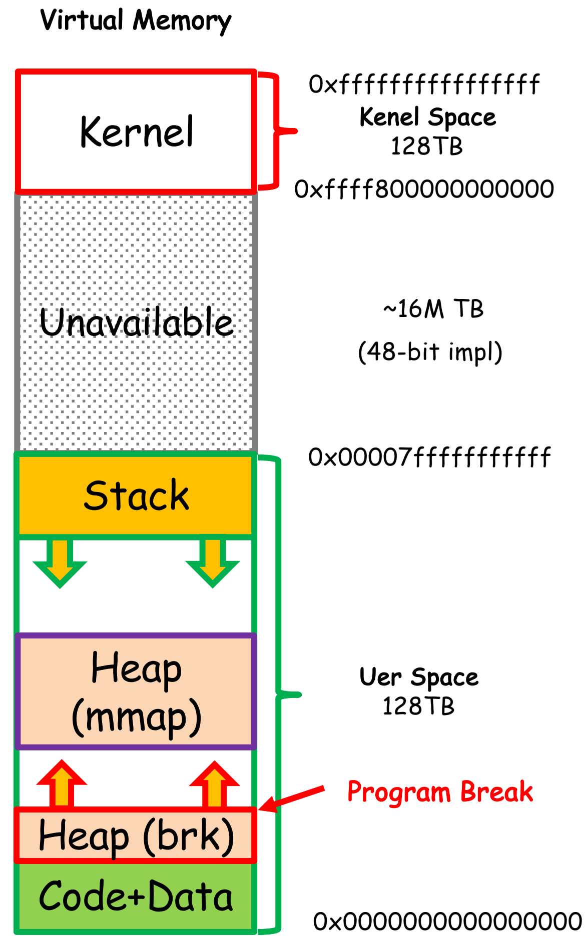
- Can you find a list to overflow the program stack?

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

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

void main(void){




    process(list, 0);
}
```

Stack Size is Limited

- The default stack size for each thread is 8MB in Linux
- Reching the limit would cause stack overflow
- Why not use a large stack?
  - Stack is mainly used to save the contexts of function calls
  - Developers should not place large data on stack
  - The remaining risk of stack overflow is deep recursion

```
#: 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 Limit

- System users: ulimit command
- Developers: use setrlimit() in their code

```
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);
```

Can Processes Know Stack Overflow?

- Yes, but usally killed by the OS directly
- We can register the SIGSEGV signal
- But executing the handler needs an extra stack
  - need to register another stack with enough space
- You will learn this in your first in-class practice

2. Heap Exhaustion

Overcommit

- When malloc() returns successful, the physical memory may not be allocated
- Linux has three options
  - 1: always overcommit, never check
  - 2: always check, never overcommit
  - 0: heuristic overcommit (this is the default)
- Try the following program with different settings

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

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

To Small to Fail & OOM Killer

- If the required space is small (usually < 8 pages), malloc() never fails when overcommit is enabled
- 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
- Try the following program

```
#define SMALL_SIZE 1024L

void main(){
    for(long i=0; i < INT64_MAX; i++) {
        char* p = malloc (SMALL_SIZE);
        if(p == 0){
            printf("the %ldth malloc failed\n", i);
            break;
        }
}
```

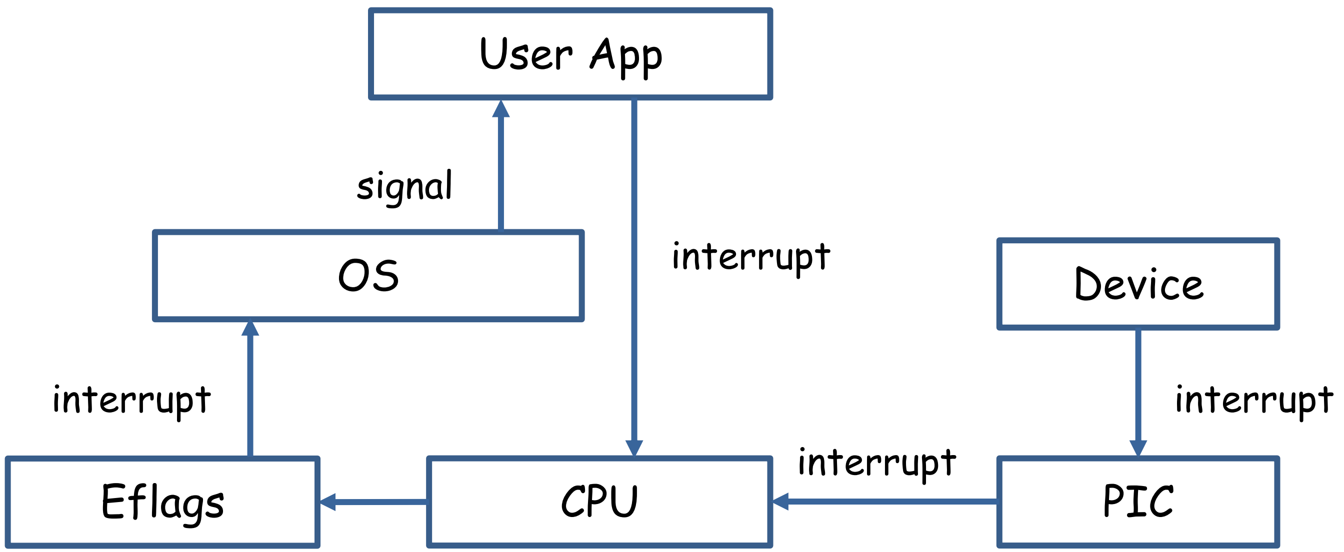
Can Processes Know Heap Exhaustion?

- malloc() returns 0 if fails
- What about to small to fail? or being killed by the OOM Killer?

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 sigaction or signal

```
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;
}
```

Exception Handling Issue

- Where should the process continue?
- How to restore the required context?
  - restore all callee-saved registers: rbp、rsp、rbx、r12-r15
- Which resource should be release?
  - heap

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");
}
```

Cleanup Attribute

- Use the attribute to set a cleanup function that should be executed before the function returns
- The function is ineffective if an exception occurs

```
void free_buffer(char **buffer){
  printf("Freeing buffer\n");
  free(*buffer);
}

void toy(){
  char *buf __attribute__ ((__cleanup__(free_buffer))) = malloc(10);
  snprintf(buf, 10, "%s", "any chars");
  printf("Buffer: %s\n", buf);
}
```

In-Class Practice 1

- Try to capture and handle the segmentation faults caused by stack overflow
  - Useful APIs: setjmp/longjmp, sigaction, sigaltstack
  - ref: https://man7.org/linux/man-pages/man2/sigaltstack.2.html

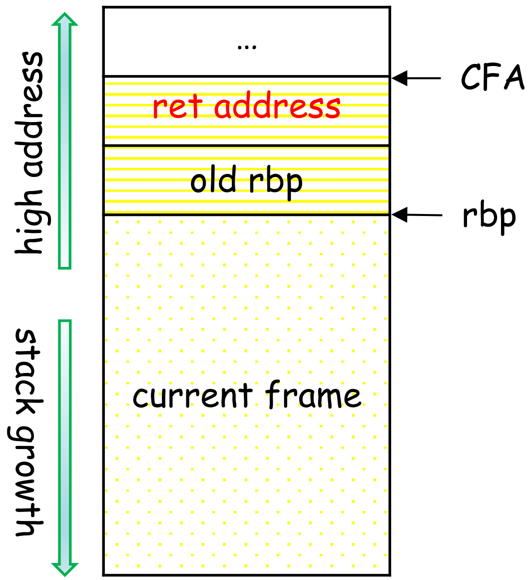
```c
#define SIGSTACK_SIZE 1024

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

void process(struct List* list, int cnt){
    if(list->next != NULL)
        process(list->next, ++cnt);
}

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

4、Stack Unwinding

Problem

- Callee-saved registers should be restored
- Setjmp is inefficient if widely used

```
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
```



Solution with 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,

mov %rsp,%rbp → 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++ throw-try-catch
  - Rust stack unwinding

In-Class Practice 2

- Calculate the DWARF info for the assembly code and fill in the following blanks

LOC	CFA	rbp	r12	r13	ra
cab0	rsp+8				
cab6	rsp+16				
cac0	rsp+24				
cac1	rsp+32				
cb03	rsp+24				
cb05	rsp+16				
cb07	rsp+8				
cb10	rsp+32				
cb1d	rsp+24				
cb25	rsp+16				
cb27	rsp+8				

```
cab0: endbr64
cab4: push    %r13
cab6: mov     %rsi, %r13
cab9: mov     $0x2e, %esi
cabe: push    %r12
cac0: push    %rbp
cac1: mov     (%rdi), r12
cac4: mov     %r12, %rdi
cac7: call    4960
cacc: mov     0x0(%r13), %r13
cad0: mov     $0x2e, %esi
cad5: mov     %rax, %rbp
cad8: mov     %r13, %rdi
cadb: call    4960
cae0: test    %rax, %rax
cae3: jz      cb10
cae5: mov     %rax, %rsi
cae8: test    %rbp, %rbp
caeb: lea     0xcd0c(%rip), %rax
caf2: cmovz   %rax, %rbp
caf6: mov     %rbp, %rdi
caf9: call    4a80
cafe: test    %eax, %eax
cb00: jz      cb1c
cb02: pop     %rbp
cb03: pop     %r12
cb05: pop     %r13
cb07: retn
cb10: lea     0xcce7(%rip), %rsi
cb17: test    %rbp, %rbp
cb1a: jnz     caf6
cb1c: pop     %rbp
cb1d: mov     %r13, %rsi
cb20: mov     %r12, %rdi
cb23: pop     %r12
cb25: pop     %r13
cb27: jmp     4a80
```

More Reference

- The "too small to fail" memory-allocation rule, https://lwn.net/Articles/627419/
- Revisiting "too small to fail”, https://lwn.net/Articles/723317/
- Exception Handling in LLVM, https://llvm.org/docs/ExceptionHandling.html
- http://itanium-cxx-abi.github.io/cxx-abi
- Théophile et al. "Reliable and fast DWARF-based stack unwinding." OOPSLA, 2019.