CS 33: Computer Organization

Dis 1B: Week 7 Discussion

Adapted from CS 33 Discussion Slides by Uen-Tao Wang

Stack Exploits

- Consider the following function

```
int foo() {
    long a = 0x7766554433221100;
    char c[16];
    gets(c);
}
```

Stack Exploits

- Disassembled code looks like this:

```
0x400528 <+0>:     push   %rbp
0x400529 <+1>:     mov    %rsp,%rbp
0x40052c <+4>:     sub    $0x20,%rsp
0x400530 <+8>:     movabs $0x7766554433221100,%rax
0x40053a <+18>:    mov    %rax,-0x8(%rbp)
0x40053e <+22>:    lea    -0x20(%rbp),%rax
0x400542 <+26>:    mov    %rax,%rdi
0x400545 <+29>:    callq  0x4003b0 <gets@plt>
0x40054a <+34>:    leaveq
0x40054b <+35>:    retq
```

Stack Exploits

- “gets” function takes a character pointer as an argument
- Then, it asks the user to input a character string where it then copies the string into the specified character pointer
- Let’s draw the stack frame of this function

Stack Exploits

- If you typed “DJ is my TA” (11 characters), 11 bytes will be occupied from c to c + 11
- What would happen if you type “DJ is my super awesome TA!” (26 characters)?
- In the C code, we only specified character array of size 16
- “gets” doesn’t care

Stack Exploits

- How can we change the value of a?

Stack Exploits

- How can we change the return address?

Stack Exploits

- What happens when we change the return address?

Stack Exploits

- How can we inject “malicious” codes?

Processes

- The most difficult thing about processes: describing them in words
- A “process” encompasses “the act of executing a flow of instructions”
- So an executable program isn't really a process, but executing a program is
- The OS maintains a list of these executions and these are what are known as “processes”

Processes

- Think of processes as having a similar level of independence as programs
- When you open a terminal to run a program, you expect it to run in isolation compared to a program running on another terminal
- This is the primary expectation placed on processes that are running in parallel

Processes

- When it comes to processes (as with programs that are running concurrently), there is no guarantee as to the order in which the programs will execute
- This leads to a complication in parallel programming which is synchronization
- Specifically, you must make sure that your program works correctly, regardless of the order in which everything is executed

Processes

- Each process thinks that it owns everything and that everything revolves around it
    - The entire addressable memory space
    - [0x000000000000, 0xFFFFFFFFFFFF]
    - All registers
    - The CPU
- In many ways, this means that using processes is simple and safe
- If I'm running process A (say Chrome), I don't have to worry about it getting in the way of process B that is also running (say Sublime Text)

Processes

- Consider two programs A and B
- At memory address 0x10, program A is storing a variable “int a”
- If B accesses memory address 0x10, of course it's not going to find A's “int a”
- Under this restriction, it's essentially impossible for a malicious process to tamper with the memory of another process

Processes

- How to make a process:
  - fork()
- The fork() function will essentially clone the clone the existing program (memory and all) and after the point at which fork is called, two processes will be running
- The original process that called fork() is considered the parent
- The new process that was created is considered the child()

Processes

- After a process's creation the only difference between the parent and child processes is that the “child” will have 0 as the return value of fork()
- The parent's **return value** for fork will be the **pid of the child process**

Processes

- A process that creates another process (via fork) is considered the “parent”
- In general, the parents have some level of control over the children processes
- All processes maintain this hierarchy
- A process can have an unbounded number of children but only one parent

Processes

```
int main()
{
    int dummy = 0;
    if(fork() == 0) {
        dummy = 1;
    } else {
        dummy = 0;
    }
}
```

Processes

```
int main()
{
    int dummy = 0;
    if(fork() == 0) {
        dummy = 1;
    } else {
        dummy = 0;
    }
}
```

- When fork() is reached the child's value for fork() is 0
- Thus, the child will execute dummy = 1
- Meanwhile, the parent's return value for fork is non-zero, therefore it will execute dummy = 0
- In both instances the “dummy” variable has the same address in memory
- However, there is no race condition or possible issue because they have the same addresses but are located in different address spaces

Processes

```
void forkParent() {
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Processes

```
void forkChildren() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Processes

- If you have multiple cores/CPUs, you can run different processes on each core
- However, you're probably running more processes than you have CPUs on your computer
- Essentially, what invariably has to happen is that the processes have to take turn sharing the CPU

Processes

- This is done via:
  - Operating System Scheduling
  - Context Switches
- When it comes to Processes (and Threads), the OS is the king. It decides which process gets the processor and which one doesn't; it decides who lives and who dies
- For instance, if Process A has been running for 10 ms. The CPU may be taken away from Process A and given it to Process B if it pleases the OS

Processes

- In order to do this, the OS has the responsibility of maintaining the state of each running process, also known as a “context”
- The “context” of a process essentially consists of all of the stuff that the process believes it owns:
  - Values of registers
  - The stack
  - Page table
  - File table
  - Essentially, the process's identity

Processes

- When the OS decides that Process A no longer amuses it and would like to perform a context switch to Process B it:
  - Saves Process A's current state/context (ie registers, stack, etc.)
  - Restores Process B's state

Processes

- What happens when a child process completes?
- Processes can be “running”, “stopped”, or “terminated”
- A completed child process is terminated, but the resources that are used to keep track of the child process still exist
- This is (and this is completely real), known as a zombie. A zombie child

Processes

- A zombie process is one who has technically terminated
  - More formal definition of exit(0): terminates the process
- However, you can't get rid of a zombie that easily; the resources that the operating system uses to manage that process continue to exist
- A process will do so unless it is “reaped”, after which its resources are reclaimed

Processes

- One of the functions that the parent has over the child is:
    - waitpid(pid_t pid, int *status, int options)
- From the parent's perspective, the default behavior is “wait for the child process with process id of 'pid' to terminate before continuing”
- If pid == -1, waitpid will wait for any one of the child processes to complete
- waitpid essentially waits for the child to die and become a zombie so that the parent can reap it of its resources

Processes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
  if(fork() == 0)
  {
    printf("a");
  }
  else
  {
    printf("b");
    waitpid(-1, NULL, 0);
  }
  printf("c");
  exit(0);
}
```

- What’s the output of this program?

Processes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
  if(fork() == 0)
  {
    printf("a");
  }
  else
  {
    printf("b");
    waitpid(-1, NULL, 0);
  }
  printf("c");
  exit(0);
}
```

- Once fork is hit, the child will print a and the parent will print b
- Afterwards, both will print c
- The parent will not print c until the child has printed a and c
- There are no other restrictions on ordering
- abcc, acbc, bacc

Threads

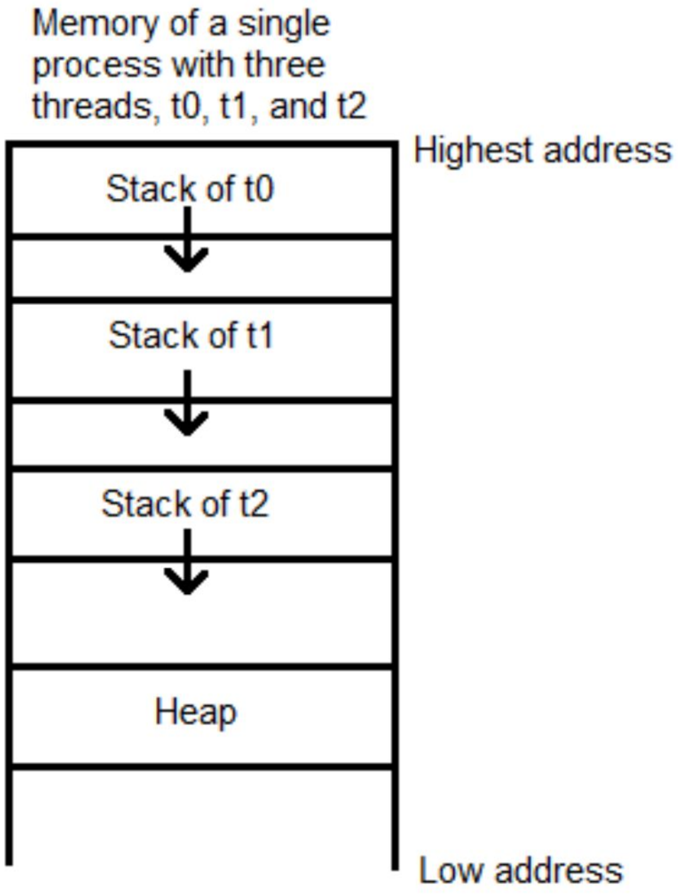
- Threads are Process' leaner, lightweight siblings
- Like processes, each thread runs its own distinct code in parallel
- However, unlike processes, threads can easily acknowledge the existence of other threads
- This is because threads share a single process and thus they share the resources of the process that they exist on

Threads

- Each thread shares with all other threads on the process the entire addressable memory space
    - [0x0000000000, 0xFFFFFFFFFF]
- Each thread has it's own space reserved in memory for its own stack
- However, each thread also believes that it is the sole owner of:
    - CPU
    - All of the registers

Threads

- Unlike with multiple processes, multiple threads on the same process share the same memory space. However, in an effort to allow for distinct execution, they each have space reserved in memory for their own stacks.



Threads

- Knowing this, what needs to be switched in a thread context switch?
  - Registers!
  - The registers include %rip and %rsp, which means switching registers will account for the different instructions that threads will execute as well as the different stacks that each thread has.

Threads

- Threads have their own stack, but memory is shared. That means that among two threads running on the same process, the address 0x10 WILL point to the same thing
- This is a major distinction from processes and can both be a major convenience or problem
- A thread know where its own stack begins and ends, but there's nothing stopping it from accessing another thread's stack

Threads

- Similar to processes, the thread's execution is at the mercy of the OS
- However, this could present a problem
- Generally when you use threads, you want them to work together in some way
- You can't assume any particular order, unless you use special functions to manually maintain order

Threads

- How to create a thread:
  - int pthread_create(pthread_t *tid, pthread_attr_t const *attr, func *f, void *args)
    - The argument tid is a thread id that is a pointer to a pthread_t passed in and assigned when the thread is created. A pthread_t is essentially a number
    - attr specifies options. By default, 0
    - This creates a thread, assigns the thread id to tid. When the thread starts, it will call f(args)

Threads

- int pthread_join(pthread_t tid, void ** thread_return)
    - One thread waits for thread tid to complete before continuing

Going back to Processes

```
#include <stdio.h>
#include <unistd.h>

int glob = 10;

void* proc_func()
{
  glob += 1;
  printf("%d\n", glob);
}
```

```
int main()
{
  fork();
  proc_func();
}
```

Threads

```
#include <stdio.h>
#include <pthread.h>

int glob = 10;

void * thread_func()
{
  glob += 1;
  printf("%d\n", glob);
}
```

```
int main()
{
  pthread_t tid1;
  pthread_t tid2;
  pthread_create(&tid1, 0, thread_func, 0);
  pthread_create(&tid2, 0, thread_func, 0);
  pthread_join(tid1, 0);
  pthread_join(tid2, 0);
}
```