CSCI 476: Computer Security

Lecture 8: Shellshock Attack (Part 1)

Reese Pearsall

Spring 2023

*all images are stolen from the internet

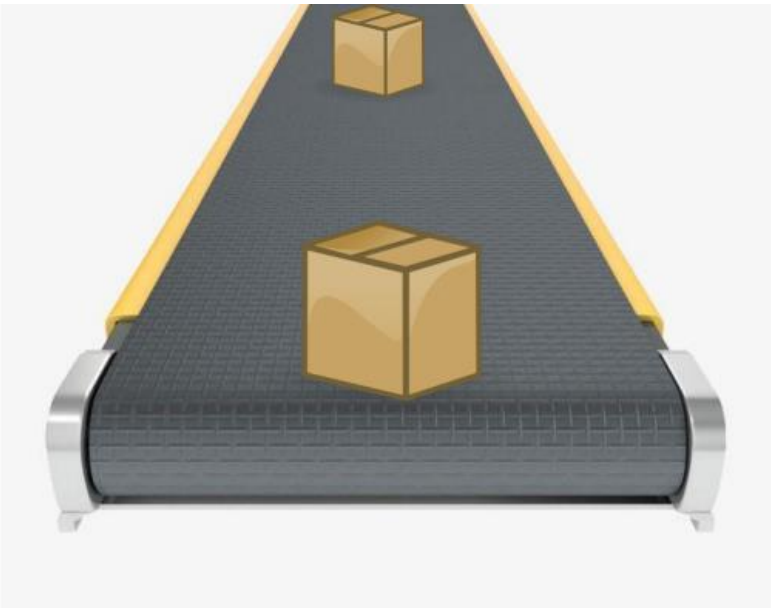
Announcements

Lab 1 (Set-UID) due on **Sunday** 2/12

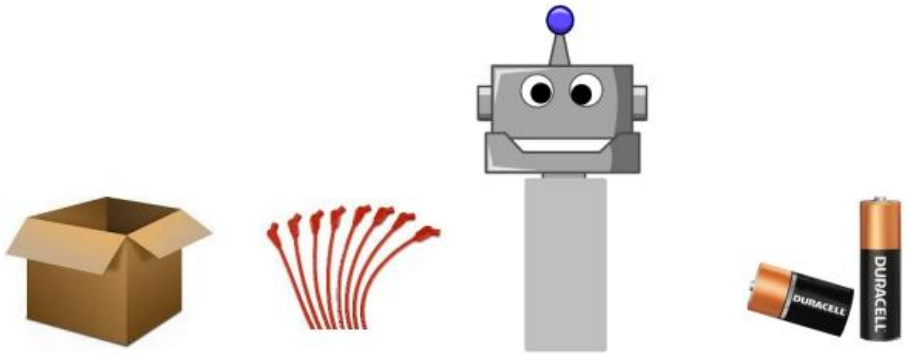
Lab 2 (Shellshock) due on **Sunday** 2/19

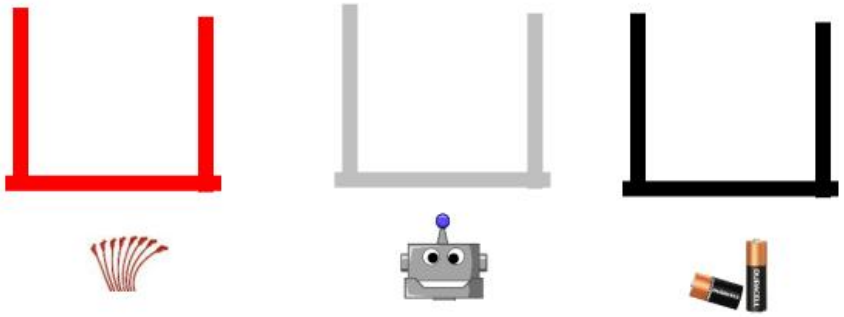
Learn about Gianforte Hall tomorrow

Factory Analogy

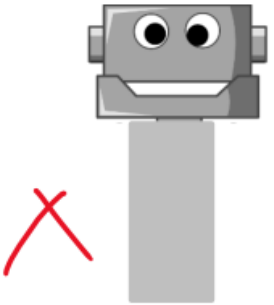


Our job is to unpack boxes we get from China, and sort the parts into the correct bins

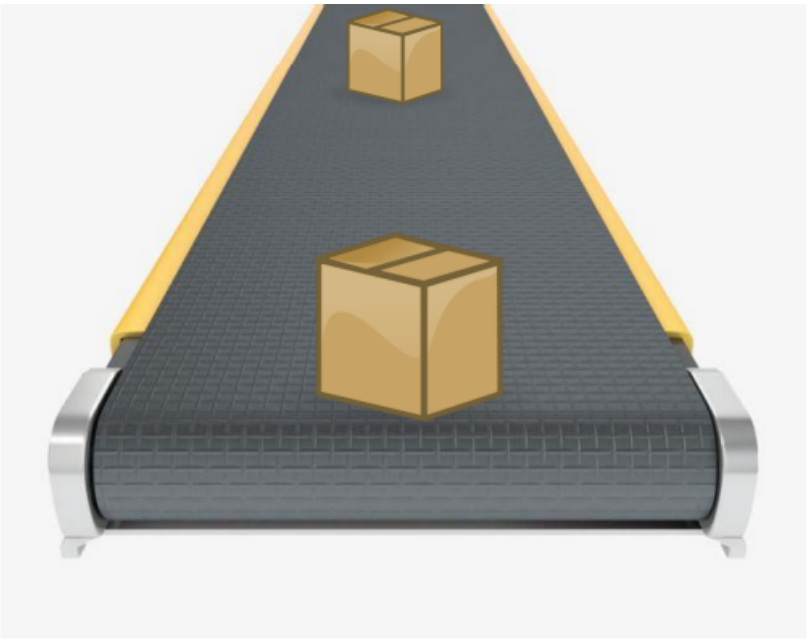




Factory Analogy

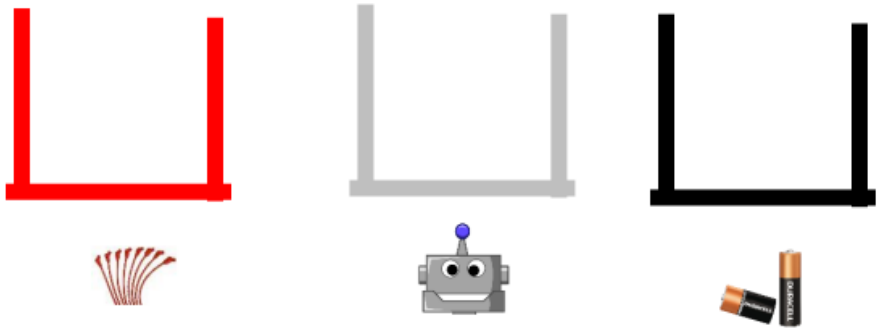


You frequently notice that the head is missing certain parts or deformed, so it should not go into the bin

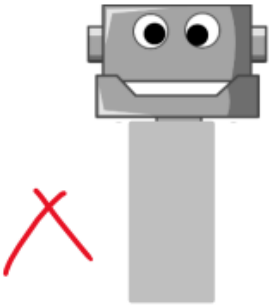


Our job is to unpack boxes we get from China, and sort the parts into the correct bins

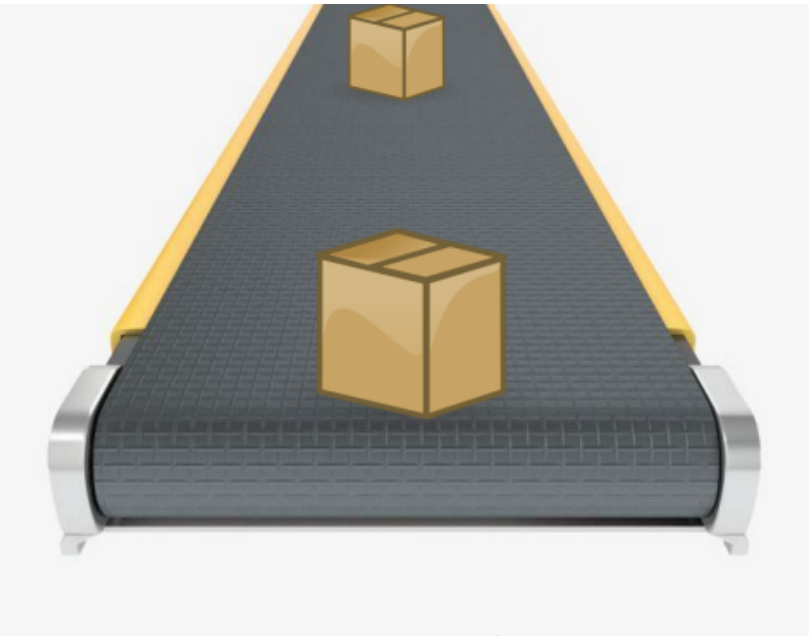
You are assigned to check the robot parts and put them in the bin



Factory Analogy

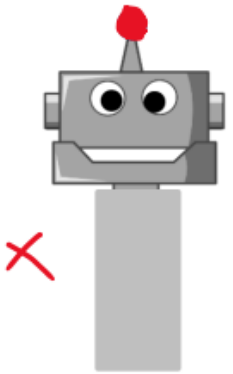


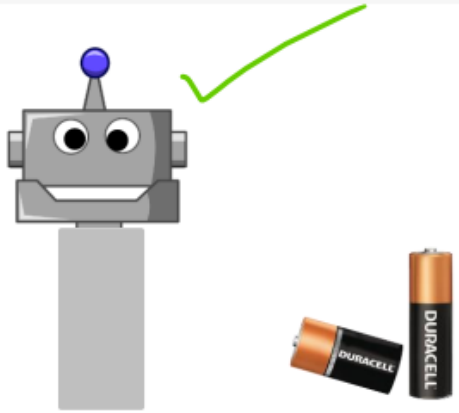
You frequently notice that the head is missing certain parts or deformed, so it should not go into the bin



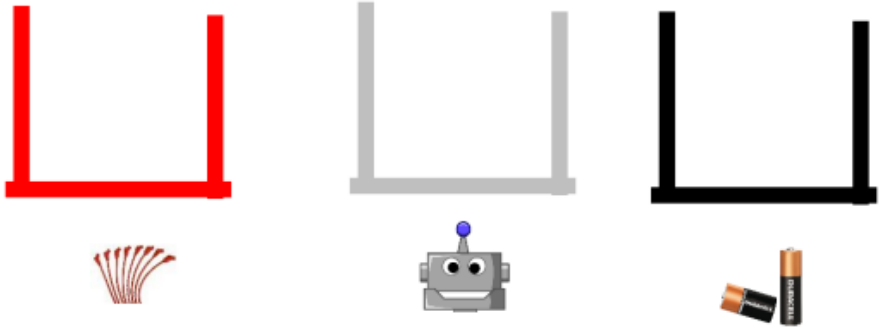
Our job is to unpack boxes we get from China, and sort the parts into the correct bins

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





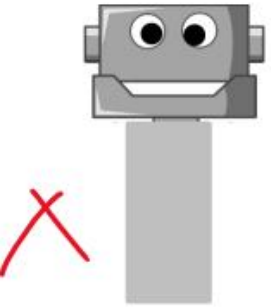
What would our program do or check for?



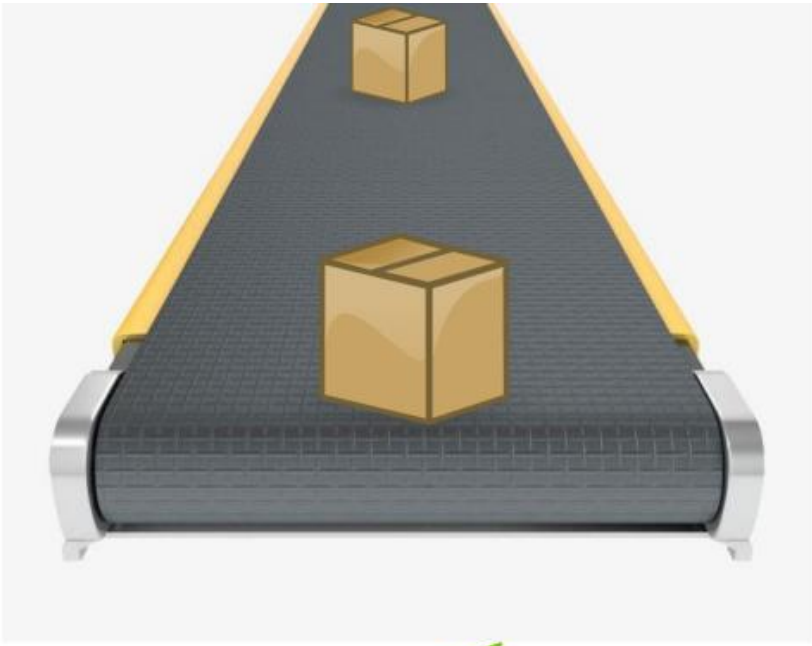
We get lazy and we write a program to check the quality of the robot for us

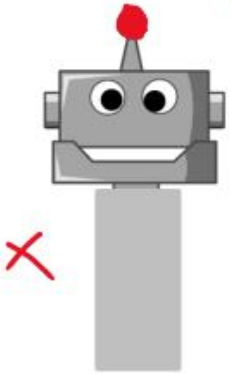
```
if (error) → throw out
else → put in bin
```

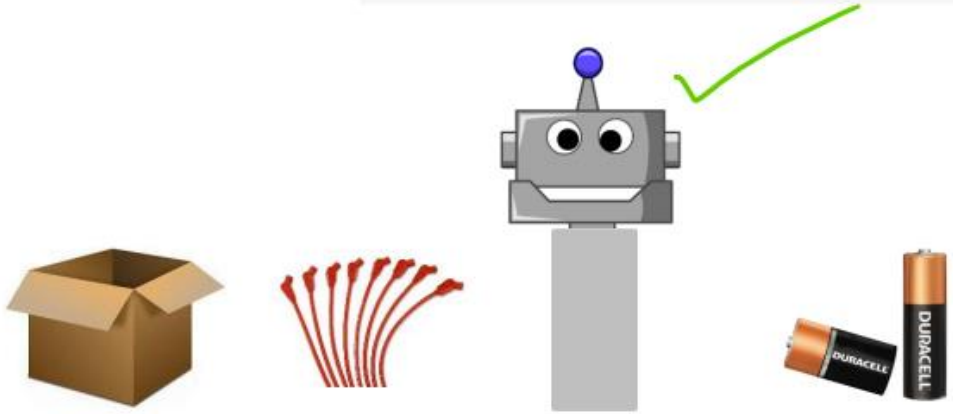
Factory Analogy



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







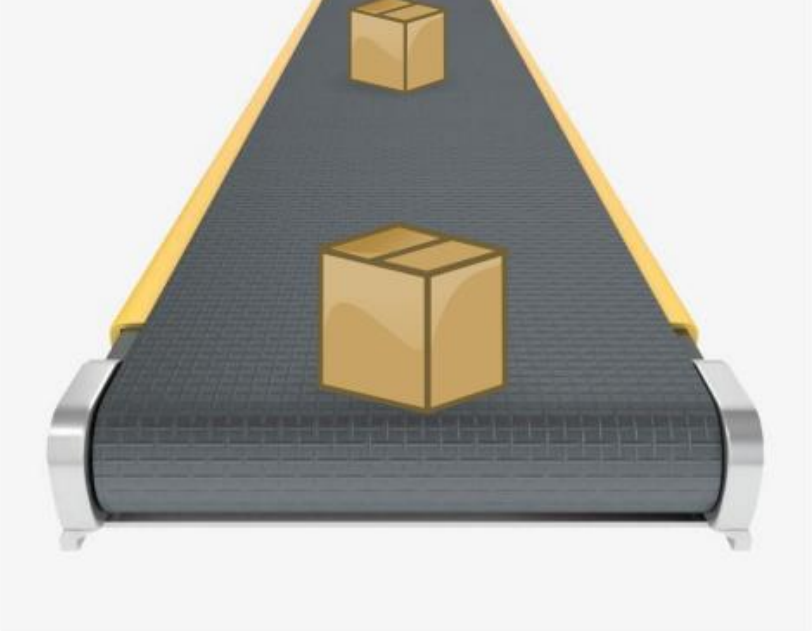
What would our program do or check for?

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


headHadErrors {

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

        return false
}
```

… right?

This will throw out any robots that has issues!

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!

Shellshock

A shell 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 Function

A **shell function** is a collection of commands to be executed under a certain name

We can use define a function a function, and declare it for use with the `declare` command

```
[02/06/23]seed@VM:~$ foo() { echo "I am a function!"; }
[02/06/23]seed@VM:~$ declare -f foo
foo ()
{
    echo "I am a function!"
}
[02/06/23]seed@VM:~$ foo
I am a function!
```

Shell Function

A **shell function** is a collection of commands to be executed under a certain name

We can also define the shell function as an **environment variable**

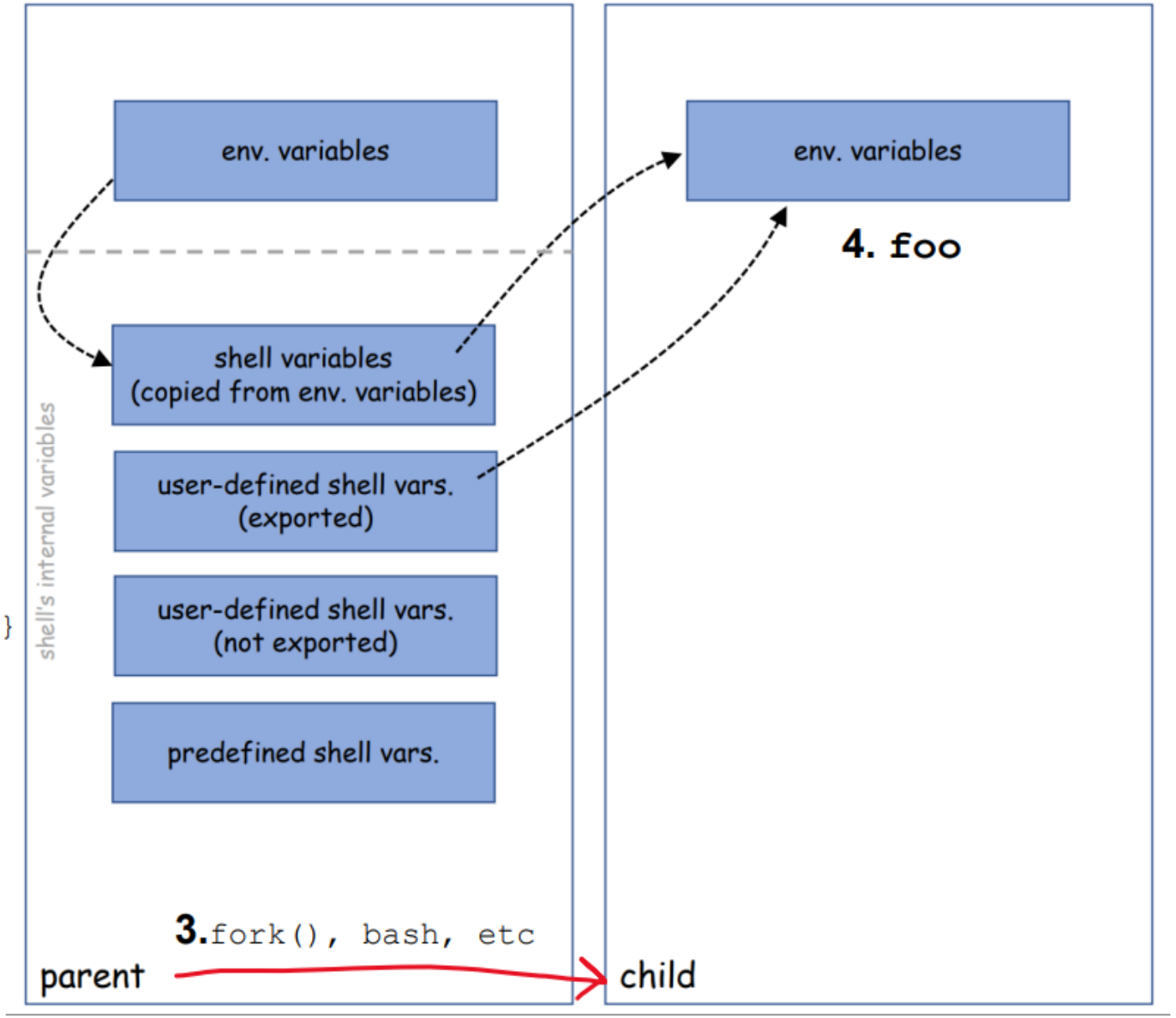
```
[02/06/23]seed@VM:~$ foo='() { echo "hello world"; }'
[02/06/23]seed@VM:~$ echo $foo
() { echo "hello world"; }
[02/06/23]seed@VM:~$ declare -f foo
[02/06/23]seed@VM:~$ export foo
```

Passing Shell Functions

Shell functions can be passed from process to process as environment variables

2. `export foo`

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



```
[02/06/23]seed@VM:~$ foo='() { echo "hello world"; }'
[02/06/23]seed@VM:~$ echo $foo
() { echo "hello world"; }
[02/06/23]seed@VM:~$ declare -f foo
[02/06/23]seed@VM:~$ export foo
[02/06/23]seed@VM:~$ bash_shellshock
[02/06/23]seed@VM:~$ echo $foo (child process)

[02/06/23]seed@VM:~$ declare -f foo (child process)
foo ()
{
    echo "hello world"
}
[02/06/23]seed@VM:~$ foo (child process)
hello world
[02/06/23]seed@VM:~$ (child process)
```

(Starts a new bash instance)

Shell functions can be passed from process to process as environment variables

In the parent process, we defined the foo function as an environment variable

In the child process, it became a shell function (no longer an env var!)

```
[02/06/23]seed@VM:~$ foo='() { echo "hello world"; }'
[02/06/23]seed@VM:~$ echo $foo
() { echo "hello world"; }
[02/06/23]seed@VM:~$ declare -f foo
[02/06/23]seed@VM:~$ export foo
[02/06/23]seed@VM:~$ bash_shellshock
[02/06/23]seed@VM:~$ echo $foo (child process)

[02/06/23]seed@VM:~$ declare -f foo (child process)
foo ()
{
    echo "hello world"
}
[02/06/23]seed@VM:~$ foo (child process)
hello world
[02/06/23]seed@VM:~$ (child process)
```

(Starts a new bash instance)

Shell functions can be passed from process to process as environment variables

In the parent process, we defined the foo function as an environment variable

In the child process, it became a shell function (no longer an env var!)

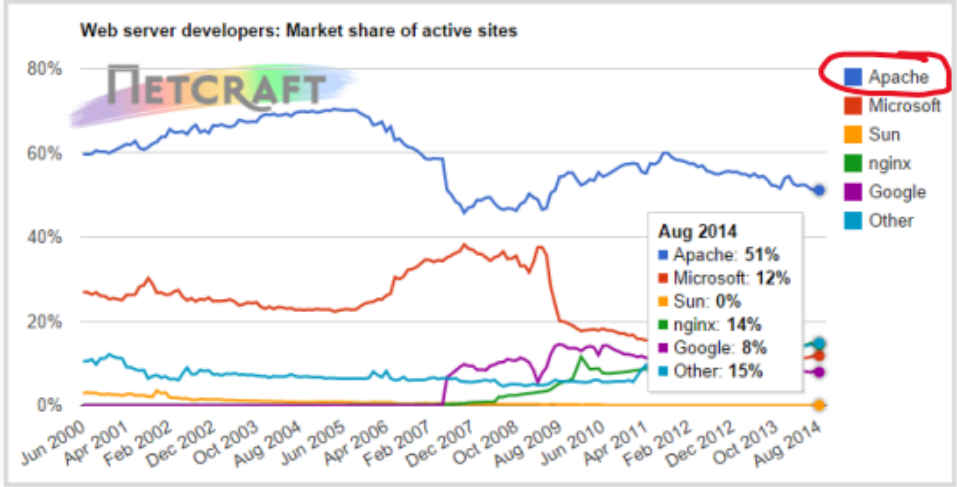
Important: Shell will parse environment variables, and if it finds a valid function definition, it will be converted to a shell function!

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 big. **Low** complexity and **high** potential damage

The *Shellshock* Vulnerability

(aka shellshock, bashbug, bashdoor)

- Disclosed Sept 24th, 2014
- 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

CVE-2014-6271: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271

Shellshock Vulnerability Demo

Shell function

An extra command

```
[02/06/23]seed@VM:~$ foo='() { echo "I am a function!"; }; echo "EVILLL";'
[02/06/23]seed@VM:~$ echo $foo
() { echo "I am a function!"; }; echo "EVILLL";
```

We define a new environment variable that has a valid function definition, but we tack on an extra command (`echo 'EVILLLL'`)

Shellshock Vulnerability Demo

Shell function

An extra command

```
[02/06/23]seed@VM:~$ foo='() { echo "I am a function!"; }; echo "EVILLL";'
[02/06/23]seed@VM:~$ echo $foo
() { echo "I am a function!"; }; echo "EVILLL";
[02/06/23]seed@VM:~$ export foo
[02/06/23]seed@VM:~$ bash shellshock
```

We export it, and then create a new bash instance (aka a new child process)

(*Remember*: environment variables get inherited by a child process)

Shellshock Vulnerability Demo

Shell function

An extra command

```
[02/06/23]seed@VM:~$ foo='() { echo "I am a function!"; }; echo "EVILLL";'
[02/06/23]seed@VM:~$ echo $foo
() { echo "I am a function!"; }; echo "EVILLL";
[02/06/23]seed@VM:~$ export foo
[02/06/23]seed@VM:~$ bash_shellshock
EVILLL
[02/06/23]seed@VM:~$ echo $foo

[02/06/23]seed@VM:~$ declare -f foo
foo ()
{
    echo "I am a function!"
}
[02/06/23]seed@VM:~$
```

!!!! Our extra command we tacked on earlier gets executed by bash!

Shellshock Vulnerability Demo

Shell function

An extra command

```
[02/06/23]seed@VM:~$ foo='() { echo "I am a function!"; }; echo "EVILLL";'
[02/06/23]seed@VM:~$ echo $foo
() { echo "I am a function!"; }; echo "EVILLL";
[02/06/23]seed@VM:~$ export foo
[02/06/23]seed@VM:~$ bash_shellshock
EVILLL
[02/06/23]seed@VM:~$ echo $foo

[02/06/23]seed@VM:~$ declare -f foo
foo ()
{
    echo "I am a function!"
}
[02/06/23]seed@VM:~$
```

!!!! Our extra command we tacked on earlier gets executed by bash!

Additionally, `foo` is now a function

Shellshock Vulnerability Demo

Shell function

An extra command

```
[02/06/23]seed@VM:~$ foo='() { echo "I am a function!"; }; echo "EVILLL";'
[02/06/23]seed@VM:~$ echo $foo
() { echo "I am a function!"; }; echo "EVILLL";
[02/06/23]seed@VM:~$ export foo
[02/06/23]seed@VM:~$ bash_shellshock
EVILLL
[02/06/23]seed@VM:~$ echo $foo

[02/06/23]seed@VM:~$ declare -f foo
foo ()
{
    echo "I am a function!"
}
[02/06/23]seed@VM:~$
```

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);

  (the rest of code is omitted)
```

If the first four characters of the environment variable are "`()  {`", then parse and execute the command(s)

The Mistake

If bash *thinks* it finds an exported function, then it calls the function `parse_and_execute()` to parse the function definition

```
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);
```

If the string contains a shell command (echo, rm, touch), then……

If bash *thinks* it finds an exported function, then it calls the function `parse_and_execute()` to parse the function definition

```
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);
```

If the string contains a shell command (echo, rm, touch), then…… **Execute it!!!**

If bash *thinks* it finds an exported function, then it calls the function `parse_and_execute()` to parse the function definition

```
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);
```

Bash only looks at first four characters, and assumes everything after that is a valid function body with no extra shell commands (bad)

The Mistake

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 Mistake

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?

Using environment variables, 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 Mistake

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"

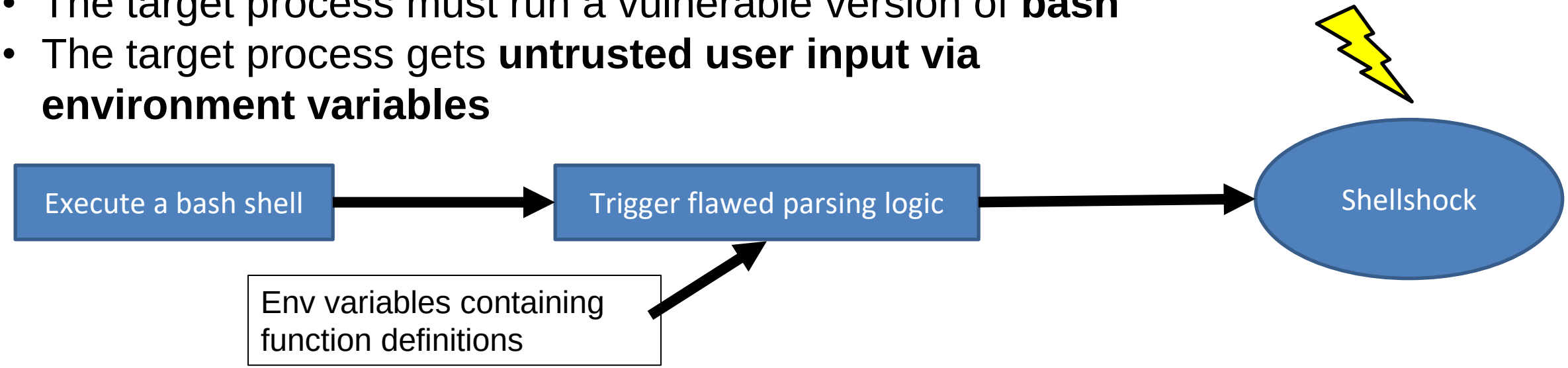
Consequences?

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

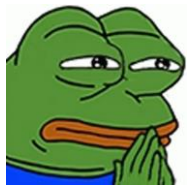
Two conditions are needed to exploit the vulnerability

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



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)

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

DO NOT try a shellshock attack on a legitimate running server

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

(In our GitHub code repository)

```
[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 *mock* server that is running the vulnerable version of bash

Folder that contains the contents for our web server

(In our GitHub code repository)

```
[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 *mock* server that is running the vulnerable version of bash

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

(In our GitHub code repository)

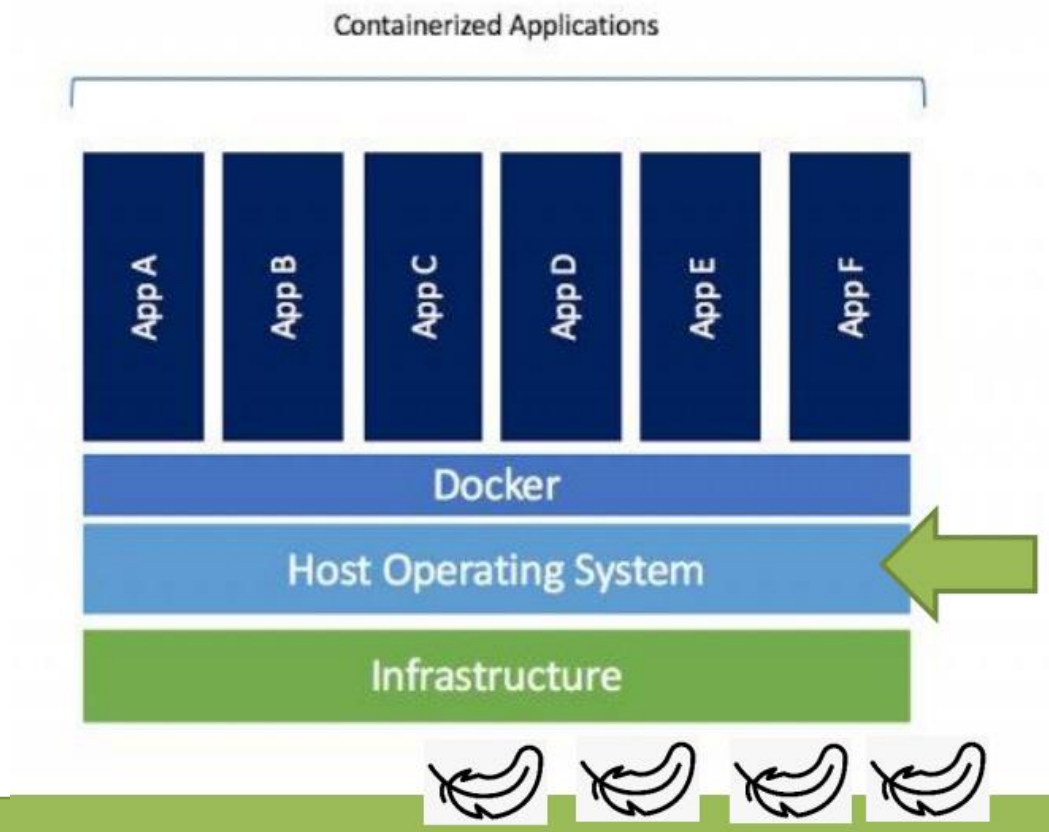
```
[09/22/22]seed@VM:~/.../02_shellshock$ ls -al
total 24
drwxrwxr-x  3 seed seed 4096 Sep 22 10:12 .
drwxrwxr-x 12 seed seed 4096 Sep 22 10:12 ..
-rw-rw-r--  1 seed seed  395 Sep 22 10:12 docker-compose.yml
drwxrwxr-x  2 seed seed 4096 Sep 22 10:12 image_www
-rw-rw-r--  1 seed seed 4430 Sep 22 10:12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


I DON'T KNOW
WHAT A
"DOCKER" IS
AND AT THIS POINT,
I'M TOO AFRAID TO ASK

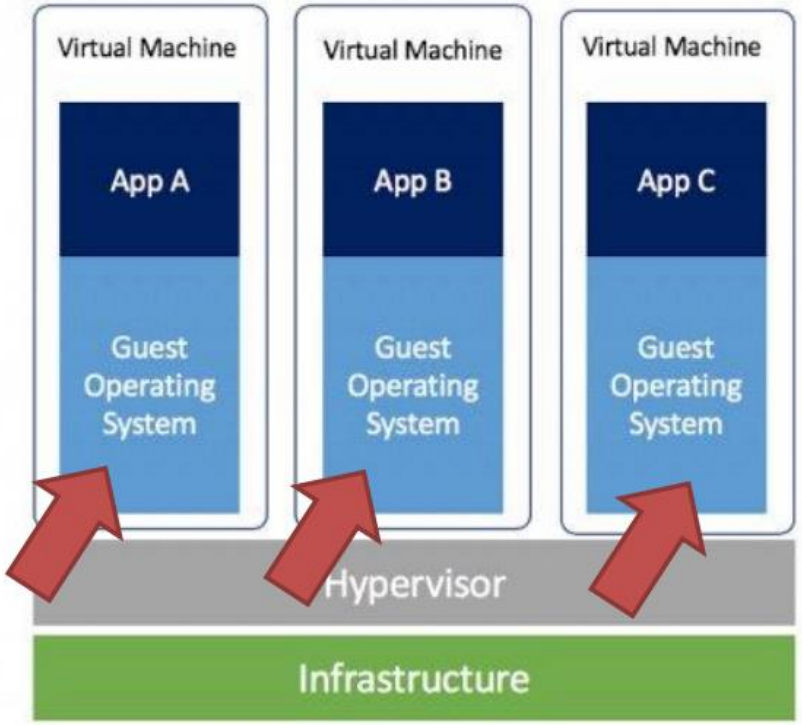
Docker

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

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



virtual machines all have their own 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)

Setting up your Docker container

```
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
```

`docker-compose up -d` is used to start the web server
(`-d` stands for “detached” and lets the web server run in the background)

```
docker-compose down          #turns the server off
```

```
docker ps -a          #view active containers and their ids
```

```
docksh <id>          #connect/log in to a container
```

```
[02/06/23]seed@VM:~/.../02_shellshock$ docker-compose up -d
Creating victim-10.9.0.80 ... done
```

```
[02/06/23]seed@VM:~/.../02_shellshock$ docker ps -a
CONTAINER ID        IMAGE                       COMMAND                  CREATED             STATUS              PORTS               NAMES
3c7859dbd392        seed-image-www-shellshock   "/bin/sh -c 'service…"   7 seconds ago       Up 6 seconds                            victim-10.9.0.80
```

```
[02/06/23]seed@VM:~/.../02_shellshock$ docksh 3c7
root@3c7859dbd392:/# whoami
root
root@3c7859dbd392:/#
```

(You don't need to provide the full container ID)

We are now logged into the docker container!