(Advanced) Computer Security!

Software Security

Shellshock

(part I)

Prof. Travis Peters

Montana State University

CS 476/594 – Computer Security

Spring 2021

https://www.traviswpeters.com/cs476

Today

Reminder!
Please update your Slack, GitHub, Zoom
(first/last name, professional photo/background)

- Announcements
  - Lab 01 – due!
  - Lab 02 - released!

- Learning Objectives
  - Tying up loose ends w/ env. vars. & set-uid programs
  - Understand shellshock and related attacks

Recap/Reflection

- What are the big ideas from Set-UID Programs? Env. Variables?

* Certain attacks are dependent on the underlying tech.
  - who uses it
  - features vs. bugs
  - ↳ Some things are more profitable to target
* need lots of info to carry out attacks (file names & locations)
* unintended consequences

* Sometimes we need to poke holes
  - ↳ be careful to make those holes small & tightly controlled

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, zsh, dash, etc.

REPL

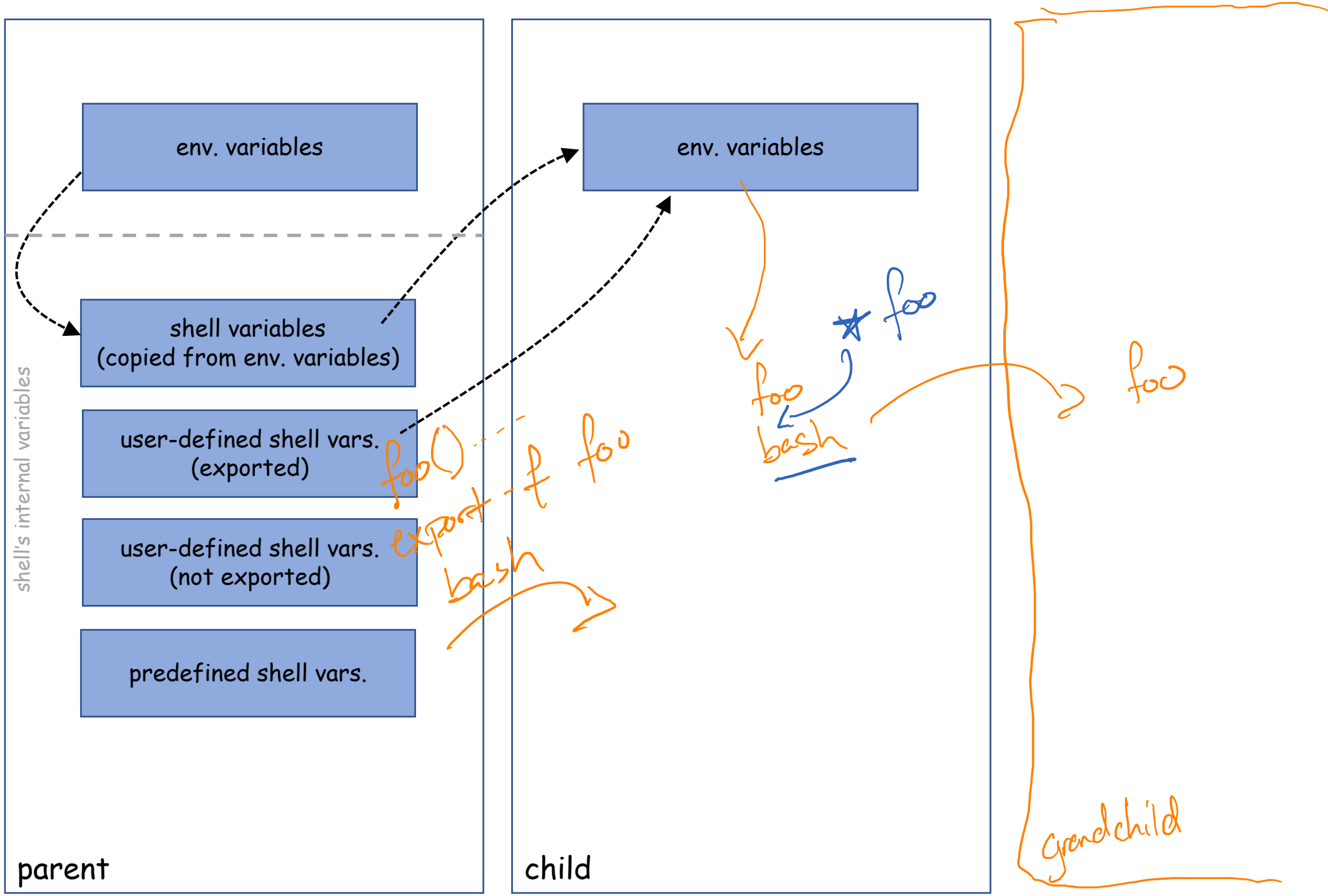
- The **bash** shell is one of the most popular shell programs; often used in the Linux OS

- 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 Vars. vs. Env. Vars



Passing Shell Functions: Parent -> Child

```
[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: Parent -> Child

```
[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!

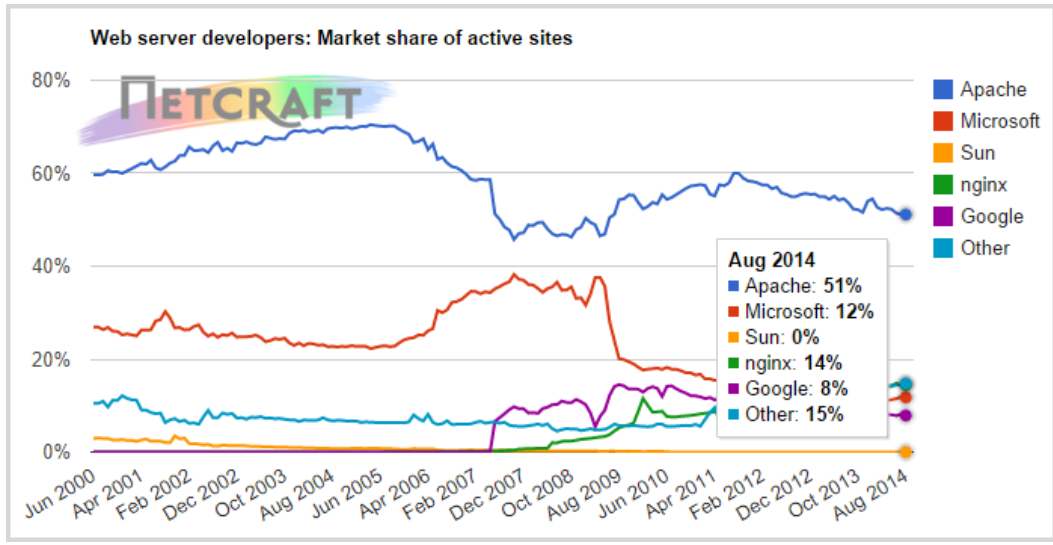
Summary: Passing Shell Functions

- 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!

- In both approaches...
  - if child process runs vuln. version of bash...
  - bash turns env. variables into function definition!



The Shellshock Vuln.

e.g., read/write files,
GET A SHELL on someone's server!

Shellshock was classified as being an **extremely critical bug** (complexity = low, potential damage = high!).

Plus, many Linux servers use bash as the default shell, and it is wayyyy too easy to find targets... mass port scanning, nmap shellshock script, metasploit module, online scanners, ...

The Shellshock Vuln.

- aka "shellshock", "bashbug", "bashdoor" – disclosed Sept. 24, 2014

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

- This vuln. exploited a mistake made by **bash** when it converts env. vars. to functions defs.
  (effectively allows remote command execution via bash)

- After the official disclosure, several other bugs were found in the bash source code.
  (Shellshock refers to the family of security bugs found in bash)

- The bug has existed in the bash source code since August 5th, 1989
  (BEFORE I WAS EVEN BORN!!!)

The Shellshock Vuln. (cont.)

- The parent process can pass a func. def. to a child shell process via env. variables
- Parsing bug -> bash **EXECUTES TRAILING COMMANDS** contained in env. variables

```
[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”
}
```

The Mistake…

- The Shellshock bug starts in the **variables.c** file in the bash source code
- The following code snippet highlights the mistake:

```
void initialize_shell_variables (env, privmode)
     char **env;
     int privmode;
{
  [...]
  for (string_index = 0; string = env[string_index++]; ) {
      [...]
      /* If exported function, define it now. ... */
      if (privmode == 0 && read_but_dont_execute == 0 && STREQN ("() {", string, 4)) {
        [...]
        parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
        [...]
      }
}
```

-> Literally, parse and execute the commands in temp_string!

Adapted from https://security.stackexchange.com/questions/68448/where-is-bash-shellshock-vulnerability-in-source-code

The Mistake... (cont.)

```
void initialize_shell_variables (env, privmode)
     char **env;
     int privmode;
{
  [...]
  for (string_index = 0; string = env[string_index++]; ) {
      [...]
      /* If exported function, define it now. ... */
      if (privmode == 0 && read_but_dont_execute == 0 && STREQN ("() {", string, 4)) {
        [...]
        parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
        [...]
      }
}
```

- At the **"if" statement**, bash checks if there is an exported function
  i.e., check whether the value of an env. variable starts with "() {" or not.
  if found, bash replaces the "=" with a space.

- Bash then calls the function **parse_and_execute(...)** to parse the func. definition
  Unfortunately, this can parse other shell commands, not just the function definition!

  - If the string is a function definition --> parse it but don't execute it

  - If the string contains a shell command(s)--> execute it

The Shellshock Vuln. (cont.)

- 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”;
```

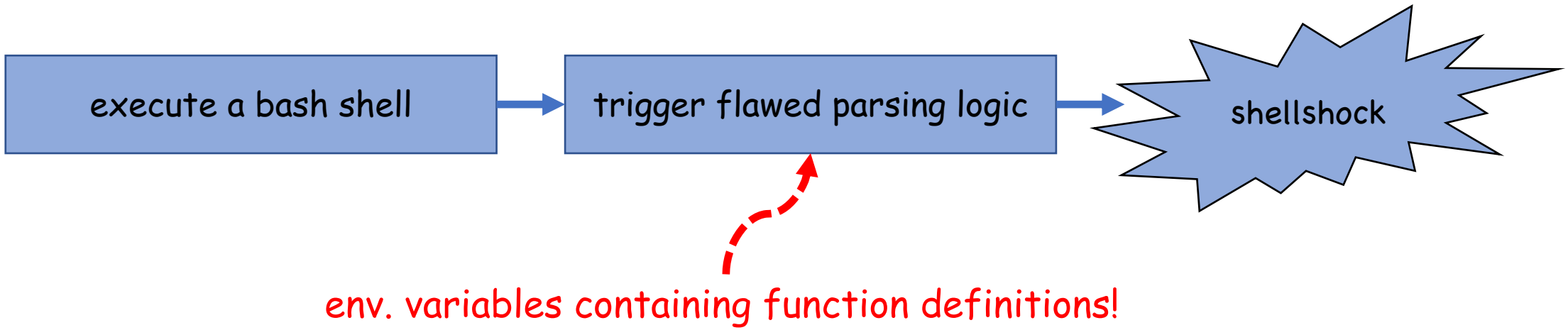
- In **B** we see that the string now becomes **two commands!**
- Now, parse_and_execute() **will execute *both* commands!**

- **Consequences**
  - Attackers can get a process to run their commands
  - If the target process is a server process or runs with elevated privileges, a security breach can occur!

(Recap) Exploiting the Shellshock Vuln.

Two conditions are needed to exploit the vulnerability:

- The target process must run (a vuln. version of) **bash**
- The target process gets **untrusted user inputs via env. variables**



(Recap) Exploiting the Shellshock Vuln.

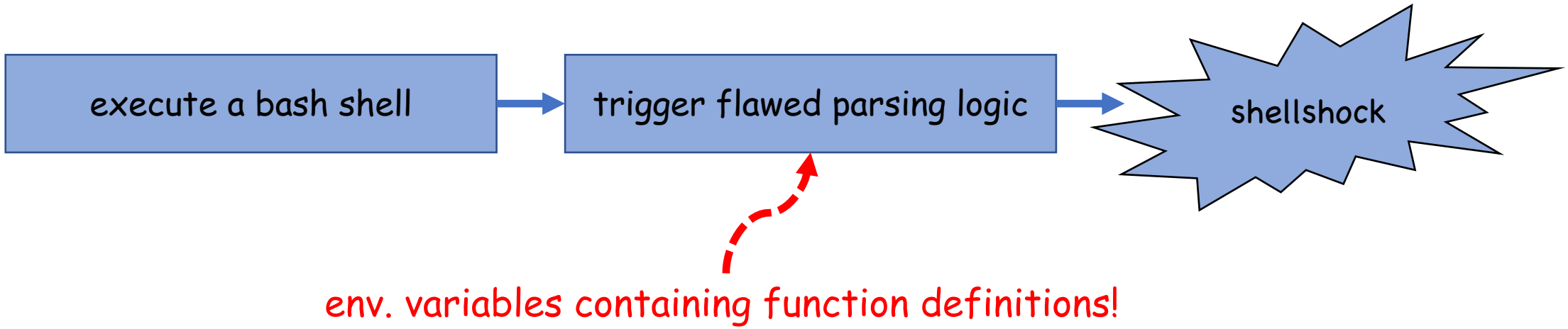
```
[1]$ foo=‘() { echo "hello world"; }; echo “extra”;’
```

Child process (/bin/bash) inherits env. vars.

```
[2]$ foo=() { echo "hello world"; }; echo “extra”;
```

Bash parsing bug! Execute trailing commands!

```
[3]$ foo () { echo "hello world"; }; echo “extra”;
```



NOTE: The Patch... one of them at least ☺

Patches are available – whether they have been applied is another thing...

```
-     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);
```

Adapted from https://security.stackexchange.com/questions/68448/where-is-bash-shellshock-vulnerability-in-source-code

New Flags/Functionality:
-> check: only allow func. definition
-> execute only one command