

fork()

Syntax: `pid = fork();`

Get almost identical copy (child) of the original (parent)

- File descriptors, arguments, memory, stack ... all copied
- Even current program counter
- But not completely identical - why?

Return value from fork call is different:

- 0 in child
- $PID > 0$ of the child when returning in parent

fork() cont.

```
pid_t child = fork();
switch (child) {
    case -1:
        //something went wrong ...
        exit(1);
    case 0:
        //I'm the child
        break;
    default:
        //I'm the parent and the child's pid is child
        break;
}
```

- `system()` wrapper around `fork()` then `exec*()`
 - Implemented in `libc.so` **not** a system call

exec()

Change program in process

- i.e., launch a new program that **replaces** the current one

Several different forms with slightly different syntax

```
int status;
```

```
status = execve(prog, args, env);
```

↑
-1 on error.
never see this
if successful

↑
name of file
that should
be executed

↑
command line
arguments ->
char* args[]

↑
environment
variables ->
char* args[]

What does execve return?

Monitoring Programs

```
pid_t waitpid(pid_t pid, int* status, int options);
```

- `wait*()` family allows parent to check status of children
 - `WIFEXITED`, `WEXITSTATUS`
 - `WIFSIGNALED`, `WTERMSIG`
 - `WIFSTOPPED`, `WSTOPSIG`
- Performing `wait*()` is **required** to clean up *zombie processes*
 - Otherwise, terminated programs remain in Z state

DEMO

The Walking Dead

Process Hierarchy

#pstree -p

```
systemd(1)-+-/usr/bin/termin(15927)-+-bash(15934)---sudo(15936)---less(15938)
|
|   |-bash(16221)-+-less(4553)
|   |   |
|   |   |-objdump(4552)
|   |   |
|   |   |-bash(21840)---pstree(4589)
|   |   |
|   |   |-bash(21925)---evince(24646)-+-{EvJobScheduler}(24663)
|   |   |   |
|   |   |   |-{dconf worker}(24653)
|   |   |   |
|   |   |   |-{gdbus}(24647)
|   |   |   |
|   |   |   `-{gmain}(24652)
|   |   |
|   |   |-bash(22574)---ssh(4333)
|   |   |
|   |   |-gnome-pty-helpe(15933)
|   |   |
|   |   |-{gdbus}(15932)
|   |   |
|   |   `-{gmain}(15935)
|   |
|   -/usr/bin/termin(27360)-+-bash(12412)
|   |
|   |   |-bash(21364)
|   |   |
|   |   |-bash(27367)
|   |   |
|   |   |-bash(27369)
|   |   |
|   |   |-bash(29751)
|   |   |
|   |   |-bash(30815)
|   |   |
|   |   |-bash(30823)
|   |   |
|   |   |-gnome-pty-helpe(27366)
```

PATH Modification

```
$ echo $PATH
/home/pizzaman/bin:/usr/local/bin:/usr/bin:/bin
$ which python
/usr/bin/python
$ ls -l /usr/bin/python
lrwxrwxrwx 1 root root 9 Jul 11 19:22 /usr/bin/python ->
python2.7
```

- Environment variables set important shell parameters
- PATH contains colon-delimited set of directories to search for commands

What happens if you can set PATH for a privileged program?

- Similar attack applies to HOME

IFS Modification

```
$ for f in blah0 blah1 blah2; do echo $f; done  
blah0 blah1 blah2
```

```
$ IFS='b'
```

```
$ for f in blah0 blah1 blah2; do echo $f; done  
lah0 lah1 lah2
```

- IFS (internal field separator) is used to parse tokens
- Classic attack is to set IFS="/"

What happens when user executes `/bin/ls`

preserve Attack

- `/usr/lib/preserve` was SUID root
- Called “`/bin/mail`” when `vi` crashed to preserve modifications to the file
- Attack
 1. Change IFS to “`/`”
 2. Create `bin` as link to `/bin/sh`
 3. Kill `vi`
 4. Profit!

Shell Injection

Shell interprets number of special characters

- ; ... Separate distinct commands
- & ... Execute in the background
- | ... Pipe output as input to another command
- > ... Redirect output to a file
- # ... Comment
- \$var ... Reference variable var
- x && y ... If x, then y
- x || y ... x or y

Shell Injection

```
$ cat vuln.sh  
#!/bin/sh  
cmd="ls $1"  
sh -c "$cmd"
```

- Injecting special characters into commands can modify intended behavior
 - Applies to command line *and* C functions that perform shell interpretation (e.g., *system()*)
- Possible whenever unsanitized, untrusted input flows to a shell invocation

DEMO
Shell Injection
bash -r

Shell Attacks

`system(char *cmd)`

- Invokes external commands via shell
- Executes `cmd` by calling `/bin/sh -c cmd`
- Can make binary program vulnerable to shell attacks
- Sanitize user input!

`popen(char *cmd, char *type)`

- Forks a process, opens a pipe, and invokes shell for `cmd`

Startup File Injection

Shells typically *source* scripts at startup

- /etc/profile, /etc/bash.bashrc,
\$HOME/.bash_profile
- \$ wc -l ~/.bashrc
115 /home/pizzaman/.bashrc

Injecting commands in startup files can be devastating

- How often do you inspect yours?

Defending Against Shell Attacks

- Restricted shells
 - Invoked using `-r`
 - Disallows SHELL, PATH, ENV modifications, `chdir`, ...
- Stripping or escaping special characters
 - `s/;|\\&|\\|...//g`
- Parsing arguments and avoiding shell interpretation
 - `execve()` instead of `system()`

DEMO
system()