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()

File Descriptor Attacks

- SUID program opens file & exec process
  - Sometimes under user control
- On-execute flag
  - If `close-on-exec` flag is not set (default), then new process inherits file descriptor
  - Avenue for attack
- Linux Perl 5.6.0
  - Perl `getpwuid()` leaves `/etc/shadow` open (June 2002)
  - Problem for Apache with `mod_perl`
- Defense: close prior to exec untrusted programs
  - Manually
  - Or, automatically using `fcntl(fd, F_SETFD, FD_CLOEXEC)`

Resource Limits

- Linux systems have built-in mechanisms for enforcing quotas
  - Hard limits can never be exceeded
  - Soft limits can be temporarily exceeded
  - Can be defined per mount point
- File system limits (quotas)
  - Restricts max allocations of storage blocks and inodes
  - `man quota`
- Process limits
  - Max # of child process, open file descriptors, etc.
- Set with `limits.conf`, `ulimit`, `setrlimit()`

ulimit -a

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 62353
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 65536
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 62353
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Resource Limits & Isolation

- Many security solutions are built on concepts of isolation and limiting access to resources
  - Virtual memory (provides isolation of memory between processes)
  - chroot (isolation between “file systems”)
  - Namespaces (isolation for many different system aspects)
  - Virtual machines (isolation between multiple OS kernels)

chroot

- Set a new root directory for a subtree of processes
- Attempts to ensure that processes cannot see “outside” of their root
- Found to be a weak security boundary, as there are many ways to circumvent it

Control Groups (cgroups)

- Limit, account for, and isolate resource usage of a collection of processes
  - CPU
  - Memory
  - disk I/O
  - network
  - etc.
- Supported by the Linux kernel since 2008

Namespaces

- Groups of processes that cannot "see" resources in other groups
  - PID
    (same PID can be used in different namespaces)
  - Network
    (multiple network stacks possible)
  - User namespaces
    (same UID can belong to different users in different namespaces)
  - Mount
  - etc.
- How to make a new namespace?
  - Ask the OS to put a process into a new namespace (i.e., system calls)

cgroups + namespaces = containers

- By combining cgroups with namespaces we can effectively isolate groups of processes from one-another
  - Docker
  - LXC (Linux containers)
  - etc.
- Remaining attack surface?
  - The host's system call interface
  - Fairly big (> 330 system calls on modern Linux)

Virtual Machines

- Attack surface of containers might be to big
- Instead run entire copies of operating systems (incl. kernels) in isolation -> Virtual Machines
- Hardware support makes it possible to run multiple kernels on the same CPU
  - 1 Virtual Machine Montior (VMM, Hypervisor)
  - Multiple guest VMs
- Remaining attack surface, limited communication channels between the guest and the hypervisor

Shared Libraries

Most programs are dynamically linked against *shared libraries*

- Collection of (related) object files
- Included into (linked) program as needed
- Form of code reuse
- Functions & data referenced through PLT, GOT

Interaction with VM copy-on-write

- Multiple processes share a single library copy
- Library pages mapped into multiple virtual address spaces from single physical copy

Check binaries with `ldd`

Shared Libraries

Static shared library

- Address binding at link-time
- Not very flexible when library changes
- Code is fast

Dynamic shared library

- Address binding at load-time
- Uses procedure linkage table (PLT) & global offset table (GOT)
- Code is slower (indirection) – but optimized
- Loading is slow (dynamic linker binds at run-time)
- Linux: `.so` Windows: `.dll` files

PLT and GOT entries are popular attack targets

- More when discussing buffer overflows

Shared Libraries

```
$ ldd /usr/bin/vim
     linux-vdso.so.1 (0x00007fffec1fe000)
     libgtk-x11-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libgtk-x11-2.0.so.0
     libgdk-x11-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libgdk-x11-2.0.so.0
     libgdk_pixbuf-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libgdk_pixbuf-2.0.so.0
     libXt.so.6 => /usr/lib/x86_64-linux-gnu/libXt.so.6 (0x00007fb0d8b0c000)
     libX11.so.6 => /usr/lib/x86_64-linux-gnu/libX11.so.6 (0x00007fb0d87c9000)
     libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fb0d84c8000)
     libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007fb0d829d000)
     libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007fb0d8079000)
     libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007fb0d7e70000)
     libgpm.so.2 => /usr/lib/x86_64-linux-gnu/libgpm.so.2 (0x00007fb0d7c69000)
     libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fb0d7a65000)
     liblua5.2.so.0 => /usr/lib/x86_64-linux-gnu/liblua5.2.so.0
     libperl.so.5.20 => /usr/lib/x86_64-linux-gnu/libperl.so.5.20
```

… 117 libraries total

Shared Libraries

Search paths

- Default `/lib`, `/usr/lib`
- Extend via `/etc/ld.so.conf[.d/*]`
- Or, `LD_LIBRARY_PATH` (environment variable)

ELF linker also allows *preloading*

- Override system library with own version
- `LD_PRELOAD` environment variable
- Possible security hazard – How so?
- Now disabled for SUID programs

Race Conditions

- Race conditions can occur if programs depend on (unguaranteed) sequence or timing of operations
  - Often arise in multithreaded or distributed systems
- TOCTTOU (time of check to time of use)
  - Security vulnerability resulting in changes in system state between predicate evaluation and use of the predicate result
  - Requires precise timing by the attacker, or use of algorithmic complexity attacks (e.g., filesystem mazes)
- Common TOCTTOU examples
  - Checking whether file can be accessed, then opening the file
  - `mktemp()` race between checking existence of temporary file and opening it

File Access TOCTTOU

Vulnerable code (setuid program)

```
1 if (access("file", R_OK)) {
2   exit(1);
3 }
4 int fd = open("file", O_RDONLY);
5 read(fd, buf, sizeof(buf));
```

Attack

```
symlink("/etc/shadow","file")
```

After program executed line 1 but before it executes line 4

Signals

Signal

- Simple form of interrupt
- Asynchronous notification
- Can happen anywhere for process in user space
- Used to deliver segfault, CTRL-C, etc.
- `kill` command

Signal handling

- Process can install signal handlers
- When no handler is present, default behavior
  - Ignore or kill process
- Possible to catch all signals except `SIGKILL (9)`

Signal Examples

`SIGSEGV`	Segmentation violation due to an invalid virtual memory access
`SIGPIPE`	Process attempts to write to an unconnected pipe or socket
`SIGALRM`	Issued when a timer elapses
`SIGSTOP`	Pauses execution of a process
`SIGKILL`	Terminates execution, cannot be caught or ignored
`SIGINT`	Interrupts process, e.g., using `CTRL-C`

Signals

- **Easy to mishandle → security issues**
  - Code must be re-entrant
  - Atomic modifications
  - No updates to global data
  - Beware of unsafe library/system calls
  - Examples

    wu-ftpd 2001, sendmail 2001/2006, stunnel 2003, ssh 2006
- **Secure signals**
  - Write handler as simple as possible
  - Block signals in handler

Debugging

- UNIX provides the `ptrace` API for debugging processes
  - Allows programs to control execution of other programs, read/write virtual memory (code and data)
- Violates process isolation, so restrictions apply
  - Must be superuser, or possess same UID
- Kernel records debugger as a special, second tracing parent process
  - Can only have *one* trace parent at any given time
  - Can be used to implement a form of evasion

Debug Evasion

Linux debugger check

```
parent = getpid()
if (!(child = fork())) {
  if (ptrace(PTRACE_ATTACH,parent,0 ,0) == -1)
    //debugger already present for parent
}
```

Windows PEB debugger check

```
mov eax, fs:[0x30]
mov eax, byte[eax+2]
test eax, eax
jne .detected_debugger
```

Questions?

END