Unix Security

Processes

Adrian Coleșa

Universitatea Tehnică din Cluj-Napoca
Computer Science Department

November 2, 2015

The purpose of this lecture

1. presents basic concepts about Unix processes and process operations
2. presents specific code vulnerabilities introduced when working with processes

Outline

Outline

1. Processes
   - Operations on Processes
   - Program Invocation
   - Process Attributes
   - File Descriptors
   - Environment Arrays
   - Shell Variables
2. Interprocess Communication
   - Pipes
   - System V IPC
   - UNIX Domain Sockets

Outline

Process Creation

- using the *fork()* system call
- the child is an identical copy of its parent
  - they share: memory, attributes, open files
  - the new process is given a new PID (process ID)
  - both processes continue their execution with the first instruction after fork
  - fork returns child PID in parent and 0 in child
- the parent-child relationship is tracked by the OS
- the child can get its parent's ID with *getppid()*
- processes whose parent terminates before them are given as their new parent the *init* process

General Parent-Child Template

```
pid_t pid;

switch (pid = fork()) {
    case -1:
      perror("fork");
      break;
    case 0:
      printf("Child: pid=getpid(), ppid=getppid()\n");
      // ... some other child job
      exit(0);
    default:
      printf("Parent: pid=getpid(), ppid=getppid()\n");
      // ... some other parent job
      exit(0);
}
```

fork() Variants

- there are some other variants of the classic fork
- *vfork()* use to avoid the performance paid for coping memory for a new process that immediately loads a new code
    - memory was shared
    - the parent is blocked until child loads other code or terminates
    - shared memory is supposed not to be changed by the child
    - as copy-on-write become common, it gets deprecated
- *rfork()* from plan9 OS is used to let the user specify the shared resources at a more granular level
- *clone()* is a similar correspondent in Linux

fork() Variants (cont.)

- they are used mainly for thread (lightweight processes) creation

Process Termination

- several ways (and reasons)
- voluntarily using *exit()*
- involuntarily, being terminated by the system by sending them signals
  - reasons: exceptions, processes sending signals, abort
  - default handling of signals is to terminate the process
  - though, some signals could be explicitly handled by the process

fork() and Open Files

- OS open file management tables
  - process file descriptor table (FDT)
  - system open file table (OFT)
  - system i-node table (IT)
- child inherits file descriptors from its parent ⇒
  - both parent's and child's FDs reference the same entries in the system OFT
  - open files are shared between parent and child ⇒
  - possible race conditions
- file descriptors can evolve independently in parent and child after fork

Outline

1. Processes
   - Operations on Processes
   - Program Invocation
   - Process Attributes
   - File Descriptors
   - Environment Arrays
   - Shell Variables
2. Interprocess Communication
   - Pipes
   - System V IPC
   - UNIX Domain Sockets

Direct Invocation

- using one function of the *exec* family
  - *execl()*, *execlp()*, *execle()*, *execv()*, *execvl()*
- the most generic one: *execve()*

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

- *argv*: command line
- *envp*: environment variables

Dangerous *execve()* Variants

- all function share the same security issues with *execve()*
- *execlp()* and *execvp* have additional concerns
- they are based on the value of PATH environment variable
- if the attacker could control the PATH, he could control which program to be loaded

Dangerous *execve()* Variants (cont.)

```
int print_directory_listing(char *path)
{
    char *argv[] = {"ls", "-l", path, NULL};
    int rc;

    rc = fork();

    if (rc < 0)
        return -1;

    if (rc == 0)
        execvp("ls", argv);
        return 0;
}
```

- setting for instance PATH to “`/tmp`” will make running a program “`/tmp/ls`”

The Argument Array

- programs usually use *switch* instruction to process their user received arguments
- some programs fail to sanitize their arguments correctly
- example (from *vacation* program): vulnerable code not sanitizing user supplied argument, which could influence the called *sendmail* program

```
void sendmessage(char *myname)
{
    ...
    if (vfork() == 0) {
      execlp(_PATH_SENDMAIL_, "sendmail", "-f", myname, from, NULL);
    }
    ...
}
```

The Argument Array (cont.)

- attack
  - send email from address "`-C/some/file/here`"
  - control *sendmail* to load an alternative configuration file
  - ⇒ execute arbitrary commands on behalf of the vacationing user
- code audit: when a program use `getopt` function be aware of
  - if the program considers option arguments in the same string with the option (like "`-C/some/file`")
  - after "`--`" the options are considered normal arguments and not handled by *getopt*

Indirect Invocation. Overview

- using functions that run a sub-shell
- specify a command line interpreted by a sub-shell
- popular functions
  - C: *system()*, *popen()*
  - Perl: *system*, *open()*
  - Java: *Runtime.getRuntime().exec()*
  - Pyhon, PHP etc.

Indirect Invocation. Security Problems

- meta-characters
  - command separators, file redirection, evaluation operators
- globbing chars for FS access
  - wildcards used to locate files based on a pattern: “.,?*[]{}”
  - inherent in shell interpreters
- environment issues
  - shell tends to change their functionality based on certain environment variables
- SUID Shell Scripts
  - generally a bad idea $\rightarrow$ can easily be tricked by meta-characters and globbing

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Process Attributes Retention

- when loading a new program (with *exec*), memory is remapped
- there are attributes inherited by the new program
- can be sources of potential vulnerabilities when
  - new application is more privileged
  - current application drops its privileges by loading the new one
- inherited attributed
  - file descriptors
  - signal mask, even if signal handlers are lost
  - effective UID / GID, except the case the new program is SUID / SGID

Process Attributes Retention (cont.)

- real UID / GID
- PID, PPID and process group ID
- supplemental groups
- working and root directory
- controlling terminal
- resource limits
- umask

Resource Limits (*rlimits*)

- enforce restrictions on the system resources that a process may use
- functions to manipulates limits: *getrlimit()* and *setrlimit()*
- each resource has two limits associated to: *soft* and *hard*
- examples of resource limits for a process
  - RLIMIT_CORE: maximum size for a core file
  - RLIMIT_CPU: maximum CPU time (sec)
  - RLIMIT_DATA: maximum size (bytes) for the data segment
  - RLIMIT_FSIZE: maximum size of a written file
  - RLIMIT_MEMLOCK: maximum no of bytes locked in memory
  - RLIMIT_NOFILE: maximum number of open files

Resource Limits (*rlimits*) (cont.)

- RLIMIT_NPROC: maximum no of processes a user can run
- RLIMIT_STACK - maximum size (bytes) for process' stack

- rlimits are useful to restrict a process

Resource Limits Vulnerabilities

- security risks
  - rlimits settings survive the *exec* calls
- attack method
  - force a called privileged process to fail in a predetermined location
- caused by
  - rlimits overrun errors not handled appropriately
  - e.g. signaled not handled
  - e.g. unfounded trusted in environment (“improbable conditions”)

Resource Limits Vulnerabilities. Example 1

```
if (!(found = !uselib(buff))) {
    if (errno != ENOENT) {
      fdprintf("2, %s: cannot load library '%s'\n", argv0, buff);
    }
}
```

- vulnerable due to
  - *buffer-overflow* in *fdprinf*
  - triggered by *overrunning rlimits*
- attack vector
  - exhaust all applications file descriptors
  - provide a long special crafted application name (argv[0])

Resource Limits Vulnerabilities. Example 2

- privileged code vulnerable to not handled RLIMIT_FSIZE

```
struct entry {
    char name[32];
    char password[256];
    struct entry *next;
};

int write_entries(FILE *fp, struct entry *list)
{
    struct entry *ent;

    for (ent = list; ent; ent=ent->next)
      fprintf(fp, "%s:%s\n", ent->name, ent->password);
    return 1;
}
```

Resource Limits Vulnerabilities. Example 2 (cont.)

- attack vector
  - set a low `RLIMIT_FSIZE`
  - mask signal `SIGXFSZ` (to be ignored) before calling the privileged program
  - could cause partial writing, e.g. truncating a password

Resource Limits Vulnerabilities. Code Audit

- check for write operations, whose result is not checked
  - both success/fail and no of written bytes
- never assume that a condition is unreachable because it seems unlikely to occur
  - rlimits could trigger such conditions by restricting resources of a privileged program

Outline

1. Processes
   - Operations on Processes
   - Program Invocation
   - Process Attributes
   - File Descriptors
   - Environment Arrays
   - Shell Variables
2. Interprocess Communication
   - Pipes
   - System V IPC
   - UNIX Domain Sockets

File Sharing Aspects

- due to *fork* or *dup* multiple fd across one or more processes refer the same open file object
    - they share all open file properties, like open mode, current position
- when multiple processes open the same file,
    - they share the same physical file and properties
    - each have a different logical view, i.e. its own open file (access mode and current position)
- an open file object and corresponding access is kept unchanged by a process even if
    - owner changed
    - permissions changed
    - file (path) removed

Close-on-Exec

- file descriptors retained over *execve()*
- UNLESS explicitly marked for closure
- setting file descriptors for *close-on-exec*
  - is a useful precaution for sensitive files to not be inherited by a subprogram
  - can be done at *open()* or with *fcntl()*
- code audit
  - for applications that creates new processes
  - check to see if there are opened files not marked for close and evaluate their implication

File Descriptor Leaks. Overview

- context
  - security checks done only at the opening of a file
  - access kept even if permissions are restricted or the application privileges are lost
- risks
  - new loaded (user controlled) code could use unintentionally inherited file descriptors
- recommendation
  - programs working with file descriptors to security-sensitive resources, should close their descriptors

File Descriptor Leaks. Examples

- vulnerable code that does not close a file descriptor to (device driver to) kernel memory

```
int kfd;
pid_t p;
char *initprog;

kfd = safe_open("/dev/kmem", O_RDWR);
init_video_mem(kfd);

if (initprog = getenv("CONTROLLER_INIT_PROGRAM")) {
    if (p=safe_fork()) { //parent
      wait_for_kid(p);
      g_controller_status = CONTROLER_READY;
    } else { //child
      drop_privs();
      execl(initprog, "conf", NULL);
      exit(0);
    }
}
```

File Descriptor Leaks. Examples (cont.)

- similar real vulnerabilities: *libkvm* (FreeBSD), *chpass* (OpenBSD)

File Descriptor Leaks. Code Audit

- programs that drop privileges to run unsafe code
  - should be evaluate from the perspective of file descriptor management
- not limited just to files
  - any resource that can be represented with a file descriptor: pipes, sockets, etc.

File Descriptor Omission. Overview

- FD allocation → the lowest available
- special (system) FDs
  - 0: STDIN
  - 1: STDOUT
  - 2: STDERR
- certain library functions consider the default associations
  - *scanf*, *gets* → `read(0, ...) //STDIN`
  - *printf*, *puts* → `write(1, ...) //STDOUT`
  - *perror* → `write(2, ...) //STDERR`
- privileged programs could be tricked to
  - write sensitive data into attacker's files
  - get inputs from attacker's files

File Descriptor Omission. Overview (cont.)

- attack vector
  - starts a SUID program with the standard file descriptors closed
  - any new file the program will open will be allocated one of the standard descriptors
  - $\Rightarrow$ program could leak important output to attacker
- fixes of such vulnerabilities
  - checking if 0, 1, and 2 are available and allocate them for "`/dev/null`"

File Descriptor Omission. Example

- vulnerable due to the possible allocation $fd = 2$

```
if ((fd = open("/etc/shadow", O_RDWR)) < 0)
  exit(1);

user = argv[1];

if ((id = find_user(fd, user)) < 0) {
  fprintf(stderr, "Error: invalid user %s\n", user);
  exit(1);
}
```

- attack
  - close(2) ⇒ the error message will go into "`/etc/shadow`"
  - username contains '`\n`' to introduce new user accounts (with root permissions) into "`/etc/shadow`"

Outline

1. Processes
   - Operations on Processes
   - Program Invocation
   - Process Attributes
   - File Descriptors
   - Environment Arrays
   - Shell Variables
2. Interprocess Communication
   - Pipes
   - System V IPC
   - UNIX Domain Sockets

Process Environment

- a collection of pairs of type "`NAME=VALUE`"
- a process' environment is maintained by the standard library
  - represented in memory as an array (*environ*) of pointers to C-like strings
  - last element is NULL
- *execve* passes the new program its environment
  - the kernel copies the environment variables into the memory of the new program at adjacent locations
  - would likely be next to program argument strings at the top of the program's stack

Process Environment (cont.)

- as a process run, it can add, modify or delete its environment variables
- new environment strings are allocated in the heap with *malloc*
- functions to manipulate the environment: *getenv*, *setenv()*, *putenv()*, *unsetenv*, *clearenv*
- the standard C library expects the strings to be in the particular format (two strings separated by '=')

Confusing *putenv()* and *setenv()*

- *putenv()* not make a copy of the string passed as argument
  - inserts the pointer directly into the environment array
  - ⇒ user can later modify data that is pointed by that pointer
  - ⇒ the pointer could be discarded
- vulnerable code: after function returns the pointer points to undefined stack data!

```
int set_editor(char *editor)
{
    char edstring[1024];

    snprintf(edstring, sizeof(estring), "EDITOR=%s", editor);

    return putenv(edstring);
}
```

Extraneous Delimiters

- variable name or value contain an extra '='
- old library functions (e.g. *setenv*, *unsetenv*) dealt differently with such cases
- current implementations normally do not accept having '=' inside the variable name
- care must be taken when an application has its own implementation for environment variable management
  - take a look at and compare how variables are found and set
  - take a look at code making assumptions the variable name contains no special delimiters

Extraneous Delimiters (cont.)

- vulnerable code: allows addition of a variable with an arbitrary value

```
int set_var(char *name)
{
    char *newenv;

    int len = strlen("APP_") + strlen("=new") + strlen("name") + 1;
    newenv = (char*) malloc(len);
    snprintf(newenv, len, "APP_%s=new", name);
    return putenv(newenv);
}
```

Duplicate Environment Variables

- have more variables with the same name defined in the environment
- current library functions are safe from this point of view
- code review: look at custom implementations
- vulnerable code: missing two consecutive entries with the same name

Duplicate Environment Variables (cont.)

```
static void _dl_undestenv(const char *var, char **env)
{
    char *ep;

    while ((ep = *env)) {
    const char *vp = var;

    while (*vp && *vp == *ep) {
        vp++;
        ep++;
    }
    if (*vp == '\0' && *ep++ == '=') {
        char **P;

        for (P=env;;++P)
        if (!(*P = *(P + 1)))
            break;
    }
    env++;
    }
}
```

Shellshock Vulnerability

- September 2014: CVE-2014-6271, CVE-2014-6277, CVE-2014-6278, CVE-2014-7169
- affected versions Bash up to 4.3
- vulnerability overview
  - environment variables whose value started with ’()’ interpreted as function definitions
  - not correctly treated: allowed extra commands to be specified after the normal function definition
- exploitation example

```
env x='() { :;}; echo vulnerable' bash -c "echo this is a test"
```

- specific exploitation methods

Shellshock Vulnerability (cont.)

- CGI-based web server
- OpenSSH server
- DHCP clients

Outline

PATH Environment Variable

- a list of directories separated by ':'
- an executable (command) name is searched in those directories
- current directory is searched only if specified explicitly
- vulnerable code: when part of a privileged application "/opt/ttt/start_process"

```
snprintf(buf, sizeof(buf),
     "/opt/ttt/logcat %s | gzcat | /opt/ttt/parse > /opt/ttt/results", logfile);
system(buf);
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

PATH Environment Variable (cont.)

- attack vector

```
$ cd /tmp
$ echo '#!/bin/sh' > gzcat
$ echo 'cp /bin/sh /tmp/sh' >> gzcat
$ echo 'chown root /tmp/sh' >> gzcat
$ echo 'chmod 4755 /tmp/sh' >> gzcat
$ chmod 755 ./gzcat
$ export PATH=.:$PATH
$ /opt/ttt/start_process
$ /tmp/sh
#
```

HOME

- indicate where the user's home directory is placed n the file system
- used in cases like "`~/file`"
- an attacker (user) can change the variable, so its good for a privileged application to check the path also in password database

IFS

- IFS = internal field separator
- tells the shell which characters represent white spaces (normally spaces, tabs, and new lines)
- if an attacker changes the IFS, it could run privileged code
- vulnerable code: not check or set IFS

```
system("/bin/ls");
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

IFS (cont.)

- attack vector

```
$ cd /tmp
$ echo 'sh -i' > bin
$ chmod 755 ./bin
$ export PATH=.:$PATH
$ export IFS='/'
$ vuln_program     # "/bin/ls" interpreted like "bin ls"
$ ./sh
#
```

- normally not working on modern shells that filters dangerous environment variables like IFS

Other Dangerous Environment Variables

- ENV (or BASH_ENV)
  - used by a non-interactive shell to run the associated filename as a startup script
  - it is usually expanded
  - attack example: `ENV=``/tmp/evil```
  - any subshells that are opened actually run the "`/tmp/evil`"
- SHELL
  - indicate the user preferred shell
- EDITOR
  - indicate the user preferred editor

Runtime Linking and Loading Variables

- LD_PRELOAD
  - provides a list of libraries that the runtime link editor loads before it loads everything else
  - gives a chance to the user to insert his own code into a process or his choosing
  - in general UNIX OSes do not honor `LD_PRELOAD` when running SUID and SGID programs
- LD_LIBRARY_PATH
  - provides a list of directories containing shared libraries
  - the runtime link editor searches through this list first when looking for shared libraries
  - it is ignored for SUID/SGID binaries

Outline

Outline

1. Processes
   - Operations on Processes
   - Program Invocation
   - Process Attributes
   - File Descriptors
   - Environment Arrays
   - Shell Variables
2. Interprocess Communication
   - Pipes
   - System V IPC
   - UNIX Domain Sockets

Anonymous Pipes

- a uni-directional pair of file descriptors
- one file descriptor for read, one for write
- created and automatically opened using *pipe()* system call
- the underlying mechanism used to run commands like "`cmd1 | cmd2`"
  - the shell creates an anonymous pipe
  - the shell creates two processes
  - the first process' STDOUT is redirected to the write file descriptor of the pipe
  - the second process' STDIN is redirected to the read file descriptor of the pipe

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Anonymous Pipes (cont.)

- first process runs first command, the second runs the second command
- *popen* system call
- writing to a pipe with no read file descriptor causes the writing program to receive a SIGPIPE signal

Named Pipes (FIFO files)

- have a name and can be opened like any other normal file
- created using *mkfifo* or *mknod*, opened with *open*
  - vulnerable at race conditions attacks, because they only create a FIFO file, but not open it
  - between creation and opening the FIFO could be replaced by an attacker
  - example: vulnerable code

```
int open_pipe(char *pipename)
{
  int rc;

  if ((rc = mkfifo(pipename, S_IRWXU)) < 0)
    return -1;
  return open(pipename, O_WRONLY);
}
```

Named Pipes (FIFO files) (cont.)

- *open* could be blocking, if mode is just for read or write only, until a counter peer process occurs
    - not a security problem in-itself, it could be used as a slowing-down (even blocking) a process in a TOCTOU attack
    - exploitation: an application opening a regular file is provided a named pipe
    - application could even be finer-tune controlled, if the attacker's application is the only writer at the other and of the pipe
- non-blocking behavior could be set explicitly
- code audit should check for the implications of

Named Pipes (FIFO files) (cont.)

- pipes created with insufficient privileges giving an attacker access to the pipe and interfering with the normal IPC
- applications intending to work with a regular file, but being provided a FIFO, because failing to determine the file type
- race conditions introduced by *mkfifo* (*mknod*) and *open*

Outline

1. Processes
   - Operations on Processes
   - Program Invocation
   - Process Attributes
   - File Descriptors
   - Environment Arrays
   - Shell Variables
2. Interprocess Communication
   - Pipes
   - System V IPC
   - UNIX Domain Sockets

Message Queues and Shared Memory

- message queues
  - a simple stateless messaging system
  - a sort of specialized message-based pipes
  - unlike file system access, message queue permissions are checked for each operation
  - functions: *msgget()*, *msgctl()*, *msgrcv()*, and *msgsnd()*
- shared memory
  - the mechanism that maps the same memory segment to more processes' address spaces
  - functions: *shmget()*, *shmctl()*, *shmat()*, and *shmdt*
- have their own namespace in kernel memory, not tied to the FS

Message Queues and Shared Memory (cont.)

- implement their own simple permissions model
- code audit: check for improper permissions of System V IPC
- after a process fork both parent and child have a copy of the mapped shared memory
- after *exec()* the shared memory is detached
- use of shared resources could introduce risks of race conditions
  - ex.: if multiple processes share the same memory, one process could change (write) some data after a process has just read (check) it

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

UNIX Domain Sockets

- similar to pipes (also anonymous and named)
- allow local processes to communicate each other
- anonymous domain sockets are created by using *socketpair()* function
  - creates a pair of unnamed endpoints that a process can use to communicate information its next children
- named sockets use the socket API functions - similar to networked applications
  - implemented using special socket device files, created automatically when a server calls *bind()*
  - location of the filename is specified in the socket address structure

UNIX Domain Sockets (cont.)

  - created with permissions "`777 & ~umask`" - exposed to attacks if "`umask = 0`"
- vulnerable code: not setting *umask* before creating the socket

```
int create_sock(char *path)
{
  struct sockaddr_un sun;
  int s;

  bzero(&sun, sizeof(sun));
  sun.sun_family = AF_UNIX;
  strncpy(sun.sun_path, path, sizeof(sun.sun_path) - 1);

  if ((s = socket(AF_UNIX, SOCK_STREAM, 0) < 0)
    return -1;

  if (bind(s, (struct sockaddr ((&sun, sizeof(sun)) < 0)
    return -1;
  return s;
}
```

UNIX Domain Sockets (cont.)

- code is also vulnerable to race conditions
  - if the user can specify parts on entirely the socket pathname
  - e.g. if user writable directories are used in the path

Bibliography

1. "The Art of Software Security Assessments", chapter 10, "Strings and Metacharacters", pp. 559 – 624