Computer Security

04. Injection Attacks (and some others)

Paul Krzyzanowski

Rutgers University

Spring 2017

Last week, we looked at

Attacks

- Buffer overflows
  - Stack overflow & return address override
  - Off-by-one overflow & frame pointer override
  - Heap overflow & data or function pointer corruption
- printf attacks
  - If you have the ability to set the format string

Last week, we looked at

- **Defenses**

- Data execution protection (DEP)
  no-execute memory pages for stack & heap
  – Attacks: return-to-libc or Return-Oriented-Programming attacks

- Address Space Layout Randomization (ASLR)
  – Attacks:
    - not all programs or libraries use ASLR
    - NOP sled – create a huge block of NOPs to increase chance of jumping to exploit
    - Try and try again if there isn't much entropy in the randomization

- Stack canaries
  – Attack: if canary is modified, the compiler causes an exception. If you can modify the exception handler, it can point to your code: *Structured Exception Handling* (SEH) exploit.

Security-Sensitive Programs

- Control hijacking isn't interesting for regular programs on your system
  - You might as well run commands from the shell
- It *is* interesting if the program
  - Has escalated privileges (setuid), especially root
  - Runs on a system you don't have access to (most servers)

Privileged programs are more sensitive & more useful targets

Injection attacks

- Injection is rated as the #1 software vulnerability in 2015 & 2016 by the Open Web Application Security Project (OWASP)

- Allows an attacker to inject code into a program or query to
  - Execute commands
  - Modify a database
  - Change data on a website

- We looked at buffer overflows and format strings
  … but there are other forms too

https://www.ibm.com/developerworks/library/se-owasptop10/

Bad Input: SQL Injection

- Let’s create an SQL query in our program

```
sprintf(buf,
    ”SELECT * WHERE user='%s' AND query='%s’;",
    uname, query);
```

- You’re careful to limit your queries to a specific user
- But suppose *query* comes from user input and is:

```
foo' OR user='root
```

- The command we create is:

```
SELECT * WHERE user='paul' AND query='foo' OR user='root’;
```

What's wrong?

- We should have used *snprintf* to avoid buffer overflow
- We didn't validate our input
  – And ended up creating a query that we did not intend to create!

Another example: password validation

- Suppose we're validating a user's password:

```
sprintf(buf,
"SELECT * from logininfo WHERE username = '%s' AND password = '%s';",
uname, passwd);
```

- But suppose the user entered this for a password:

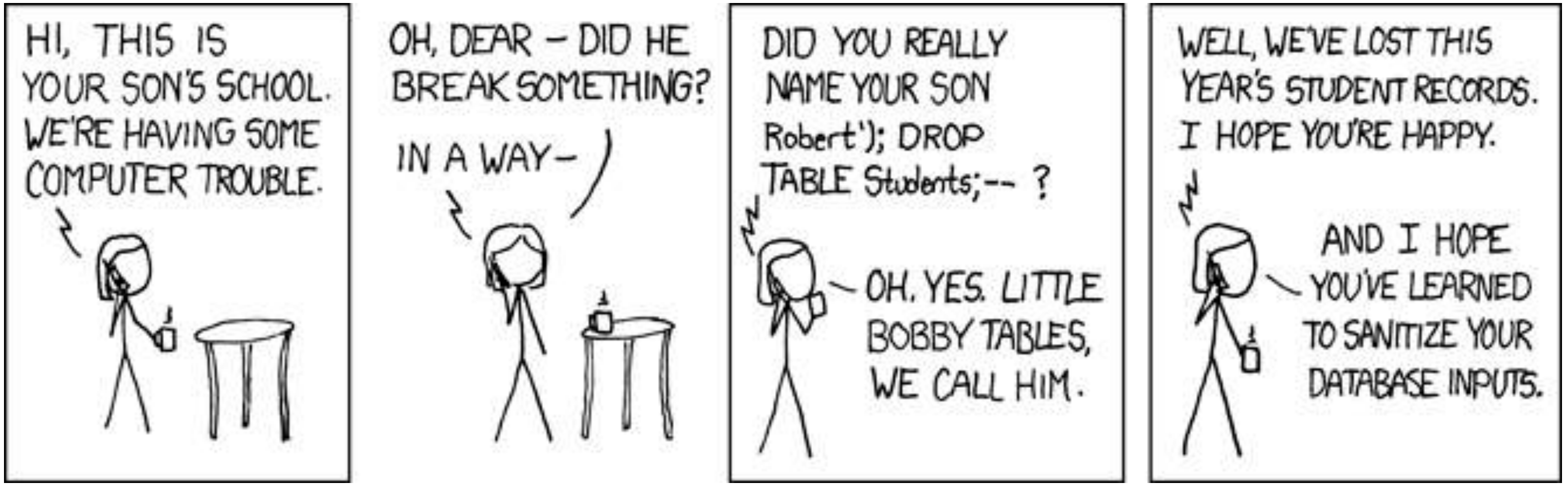
```
' OR 1=1 --
```

The -- is a comment that blocks the rest of the query (if there was more…)

- The command we create is:

```
SELECT * from logininfo WHERE username = paul AND
password = '' OR 1=1 -- ;
```

Opportunities for destructive operations



https://xkcd.com/327/

Most databases support a batched SQL statement: multiple statements separated by a semicolon

```
SELECT * FROM students WHERE name = 'Robert';DROP TABLE Students; --
```

Protection from SQL Injection

- SQL injection attacks are incredibly common because most web services are front ends to database systems
  - Input from web forms becomes part of the command
- Type checking is difficult
  - SQL contains too many words and symbols that may be legitimate in other contexts
  - Use escaping for special characters
    - Replace single quotes with two single quotes
    - Prepend backslashes for embedded potentially dangerous characters (newlines, returns, nuls
  - Escaping is error-prone
    - Rules differ for different databases (MySQL, PostgreSQL, dashDB, SQL Server, …

Don't create commands with user substrings added into them

Protection from SQL Injection

- Use parameterized SQL queries or stored procedures
  - Keeps query consistent: parameter data never becomes part of the query string

```
uname = getResourceString("username");
passwd = getResourceString("password");
query = "SELECT * FROM users WHERE username = @0 AND password = @1";
db.Execute(query, uname, passwd);
```

General Rule

- If you invoke *any* external program, know its parsing rules
- Converting data to statements that get executed is common in some interpreted languages
  - Shell, Perl, PHP, Python

IFS

Shell variable IFS (Internal Field Separator) defines delimiters used in parsing arguments

- If you can change IFS, you may change how the shell parses data
- The default is space, tab, newline

try1.sh

```
#!/bin/bash
while read name password; do
    echo name=\"$name\", password=\"$password\"
done
```

names

```
james password
mary 123456
john qwerty
patricia letmein
robert shadow
jennifer harley
```

output

```
$ ./try1.sh <names
name="james", password="password"
name="mary", password="123456"
name="john", password="qwerty"
name="patricia", password="letmein"
name="robert", password="shadow"
name="jennifer", password="harley"
```

IFS

One small change: IFS=+

try1.sh

```
#!/bin/bash
IFS=+
while read name password; do
    echo name=\"$name\", password=\"$password\"
done
```

names

```
james password
mary 123456
john qwerty
patricia letmein
robert shadow
jennifer harley
```

output

```
$ ./try1.sh <names
name="james password", password=""
name="mary 123456", password=""
name="john qwerty", password=""
name="patricia letmein", password=""
name="robert shadow", password=""
name="jennifer harley", password=""
```

IFS

It gets tricky for output

try.sh

```
#!/bin/bash

IFS='+'

echo '"$@" expansion'
echo "$@"

echo '"$*" expansion'
echo "$*"
```

```
$ ./try.sh sleepy sneezy grumpy dopey doc
"$@" expansion
sleepy sneezy grumpy dopey doc
"$*" expansion
sleepy+sneezy+grumpy+dopey+doc
```

You really have to know what you're dealing with!

Suppose a program wants to send mail. It might call:

```
FILE *fp = popen("/usr/bin/mail –s subject user", "w")
```

If IFS is set to `"/"` then the shell will try to execute usr bin mail…
An attacker needs to plant a program named "usr" anywhere in the search path

system() and popen()

- These command make it easy to execute programs
  - system: execute a shell command
  - popen: execute a shell command and get a file descriptor to send output to the command or read input from the command
- These both run sh -c
- Vulnerabilities include
  - Altering the search path if the full path is not specified
  - Changing IFS to change the definition of separators
  - Using user input as part of the command

```
snprintf(cmd, "/usr/bin/mail -s alert %s", bsize, user);
f = popen(cmd, "w");
```

What if user = `"paul;rm -fr /home/*"`

```
sh -c "/usr/bin/mail -s alert paul; rm –fr /home/*"
```

Other environment variables

- PATH: search path for commands
  - If untrusted directories are in the search path before trusted ones (/bin, /usr/bin), you might execute a command there.
    - Users sometimes place the current directory (.) at the start of their search path
    - What if the command is a booby-trap
  - If shell scripts use commands, they're vulnerable to the user's path settings
  - Use absolute paths in commands or set PATH explicitly in a script
- ENV, BASH_ENV
  - Contains a file some shells execute when a shell starts

Other environment variables

LD_LIBRARY_PATH

- Search path for shared libraries
- If you change this, you can replace parts of the C library by custom versions
  - Redefine system calls, printf, whatever…

LD_PRELOAD

- Forces a list of libraries to be loaded for a program, even if the program does not ask for them
- If we preload our libraries, they get used instead of standard ones

You won't get root access with this but you can change the behavior of programs

- Change random numbers, key generation, time-related functions in games
- List files or network connections that a program does
- Modify features or behavior of a program

Example of LD_PRELOAD

random,.c

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char **argv)
{
  int i;

  srand(time(NULL));
  for (i=0; i < 10; i++)
    printf("%d\n", rand()%100);
  return 0;
}
```

```
$ cc -o random random.c
$ ./random
9
57
13
1
83
86
45
63
51
5
```

Let's create a replacement for rand()

rand.c

```
int rand() {
    return 42;
}
```

```
$ gcc -shared -fPIC rand.c -o newrandom.so    # compile
$ export LD_PRELOAD=$PWD/newrandom.so  # preload
$ ./random
42
42
42
42
42
42
42
42
42
42
```

We didn't have to recompile *random!*

File Desciptors

- On POSIX systems
  - File descriptor 0 = *stdin*
  - File descriptor 1 = *stdout*
  - File descriptor 2 = *stderr*

- open() returns the first available file descriptor

- Vulnerability
  - Suppose you close fd 1
  - Invoke a setuid root program that will open some sensitive file for output
  - Anything the program prints to *stdout* (e.g., via *printf*) will write into that file, corrupting it.

File Descriptors - example

files.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char **argv)
{
  int fd = open("secretfile", O_WRONLY|O_CREAT, 0600);

  fprintf(stderr, "fd = %d\n", fd);
  printf("hello!\n");
  fflush(stdout); close(fd);
  return 0;
}
```

```
$ ./files
fd = 3
hello!
$ ./files >&-
fd = 1
```

Bash command to close a file descriptor

Obscurity

Windows CreateProcess function

```
BOOL WINAPI CreateProcess(
  _In_opt_     LPCTSTR               lpApplicationName,
  _Inout_opt_  LPTSTR                lpCommandLine,
  _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_         BOOL                  bInheritHandles,
  _In_         DWORD                 dwCreationFlags,
  _In_opt_     LPVOID                lpEnvironment,
  _In_opt_     LPCTSTR               lpCurrentDirectory,
  _In_         LPSTARTUPINFO         lpStartupInfo,
  _Out_        LPPROCESS_INFORMATION lpProcessInformation);
```

- 10 parameters that define window creation, security attributes, file inheritance, and others…
- It gives you a lot of control but do most programmers know what they're doing?

App-level access control: filenames

- If we allow users to supply filenames, we need to check them
- App admin may specify acceptable pathnames & directories
- Parsing is tricky
  - Particularly if wildcards are permitted (*, ?)
  - And if subdirectories are permitted

Parsing directories

- Suppose you want to restrict access outside a specified directory
  - Example, ensure a web server stays within /home/httpd/html
- Attackers might want to get other files
  - They'll put `..` in the pathnaame
  - `..` is a link to the parent directory
  - For example:
    `http://poopybrain.com/../../../etc/passwd`
  - The .. Does not have to be at the start of the name – could be anywhere
    `http://poopybrain.com/419/notes/../../416/../../../../etc/passwd`
  - But you can't just search for .. because embedded `..` is valid
    `http://poopybrain.com/419/notes/some..junk..goes..here/`
  - Also, extra slashes are fine
    `http://poopybrain.com/419////notes///some..junk..goes..here///`

Basically, it's easy to make mistakes!

Application-Specific Syntax: Unicode

Here's what Microsoft IIS did

- Checked URLs to make sure the request did not use `../` to get outside the *inetpub* web folder
  - Prevents http://www.poopybrain.com/scripts/../../winnt/system32/cmd.exe
- Then it passed the URL through a decode routine to decode extended Unicode characters
- Then it processed the web request

Application-Specific Syntax: Unicode

- What's the problem?
  - / could be encoded as unicode **%c0%af**
- UTF-8
  - If the first bit is a 0, we have a one-byte ASCII character
    - Range 0..127
  - If the first bit is 1, we have a multi-byte character
    - If the leading bits are 110, we have a 2-byte character
    - If the leading bits are 1110, we have a 3-byte character, and so on…

    / = 47 = 0x2f
  - 2-byte Unicode is in the form `110a bcde 10fg hijk`
    - 11 bits for the character # (codepoint), range 0 .. 2047
    - C0 = 1100 0000, AF = 1010 1111 which represents 0x2f = 47
  - Technically, two-byte characters should not process # < 128
    - … but programmers are sloppy … and we want the code to be fast

Application-Specific Syntax: Unicode

- Parsing ignored %c0%af as / because it shouldn't have been one
- So IIS could access ANY file in the system
- IIS ran under an IUSR account
  - Anonymous account used by IIS to access the system
  - IUSER is a member of *Everyone* and *Users* groups
  - Has access to execute most system files, including cmd.exe and command.com
- A malicious user had the ability to execute <u>*any*</u> commands on the web server
  - Delete files, create new network connections

Parsing escaped characters

- Even after Microsoft fixed the Unicode bug, another problem came up
- If you encoded the backslash (\) character
  (Microsoft uses backslashes for filenames & accepts either in URLs
- … and then encoded the encoded version of the \, you could bypass the security check

  \ = %5c
  - % = %25
  - 5 = %35
  - c = %63

  For example, we can also write:
  - %%35c => %5c => \
  - %25%35%63 => %5c => \
  - %255c => %5c => \

Yuck!

http://help.sap.com/SAPHELP_NWPI71/helpdata/en/df/c36a376a3a43ceaaa879ab726f0ec8/content.htm

These are application problems

- The OS uses what the application gives it
  - It traverses the directory tree and checks access rights as it goes along
    - "x" (search) permissions in directories
    - Read or write permissions for the file
- The application is trying to parse a pathname and map it onto a subtree

More Unicode issues

- Unicode represents virtually all the worlds glyphs
- Some symbols look the same (or similar) but have different values
  - / = solidus (slash) = U+002F
  - ⁄ = fraction slash = U+2044
  - ∕ = division slash = U+2215
  - ̷ = combining short solidus overlay = U+0337
  - ̸ = combining long solidus overlay = U+0338
  - ／ = fullwidth solidus = U+FF0F
- Like the slash, other characters may have multiple representations
  - á = U+00C1 = U+0041,U+0301
- Comparison rules have to be application dependent

Yuck!

Homograph Attack

- Some characters may look alike:
  - 1 (one), l (L), I (i)
  - 0 (zero), O
- Homograph attack = deception
  - paypal.com vs. paypaI.com (I instead of L)
- It got worse with internationalized domain names (IDN)
  - wikipedia.org
    - Cyrillic a (U+0430), e (U+435), p (U+0440)
    - Belarusian-Ukranian i (U+0456)
  - Paypal
    - Cyrillic P, a, y, p, a; ASCII l

https://en.wikipedia.org/wiki/IDN_homograph_attack

Setuid file access

- Some commands may need to write to restricted directories or files but also access user's files
- Example: some versions of lpr (print spooler)
  - Read users' files and write them to the spool directory
- Let's run the program as setuid to root
  - But we will check file permissions first to make sure the file has read access

```
if (access(file, R_OK) == 0) {
    fd = open(file, O_RDONLY);
    ret = read(fd, buf, sizeof buf);
    ...
}
else {
    perror(file);
    return -1;
}
```

Problem: TOCTTOU

```
if (access(file, R_OK) == 0) {

    fd = open(file, O_RDONLY);
    ret = read(fd, buf, sizeof buf);
    ...
}
else {
    perror(file);
    return -1;
}
```

- Race condition:
  TOCTTOU: Time of Check to Time of Use
- Window of time between *access* check & *open*
  - Attacker can create a link to a readable file
  - Run *lpr* in the background
  - Remove the link and replace it with a link to the protected file
  - The protected file will get printed

mktemp is also affected by this race condition

- Create a temporary file to store received data

```
if (tmpnam_r(filename)) {
  FILE* tmp = fopen(filename, "wb+");
  while((recv(sock, recvbuf, DATA_SIZE, 0) > 0) && (amt != 0))
    amt = fwrite(recvbuf, 1, DATA_SIZE, tmp);
}
```

- API functions to create a temporary filename
  - C library: *tmpnam, tempnam, mktemp*
  - C++: *_tempnam, _tempnam, _mktemp*
  - Windows API: *GetTempFileName*
- They create a unique name when called
  - But no guarantee that an attacker doesn't create the same name before the filename is used
  - Name often isn't very random: high chance of attacker constructing it

From https://www.owasp.org/index.php/Insecure_Temporary_File

mktemp is also affected by this race condition

- They create a unique name when called
  - But no guarantee that an attacker doesn't create the same name before the filename is used
  - Name often isn't very random: high chance of attacker constructing it
- If an attacker creates that file:
  - Access permissions may remain unchanged
    - Attacker may access the file later and read its contents
  - Legitimate code may append content, leaving attacker's content in place
    - Which may be read later as legitimate content
  - Attacker may create the file as a link to an important file
    - The application may end up corrupting that file
  - The program may be smart and call *open* with O_CREAT | O_EXCL
    - Or, in Windows: CreateFile with the CREATE_NEW attribute
    - Create a new file with exclusive access
    - But if the attacker creates a file with that name, the *open* will fail
      - Denial of service

From https://www.owasp.org/index.php/Insecure_Temporary_File

Defense against mktemp attacks

- Use *mkstemp*
- It will attempt to create & open a unique file
- You supply a template
  - A name of your choosing with XXXXXX that will be replaced to make the name unique
  - mkstemp("/tmp/secretfileXXXXXX")
- File is opened with mode 0600: r-- --- ---
- If unable to create a file, it will fail and return -1
  - You should test for failure and be prepared to work around it.

The main problem: interaction

- To increase security, a program must minimize interactions with the outside
  - Users, files, sockets
- All interactions may be attack targets
- Must be controlled, inspected, monitored

Relative Attack Surface Quotient (RASQ)

- Microsoft metric of application vulnerability
  - Attempts to mathematically quantify the attackability of software
- Roughly, measures # of input channels
  - Some channels are easier to exploit
  - Some channels are more accessible to others
- Sum of "effective attack surface values" for all "root attack vectors"
  - Root attack vector: feature that can positively or negatively affect the security of a product
  - Attack bias: value representing risk of compromise for an attack
    - Subjective measure: 0=no threat, 1=maximum threat
  - Attackable surface: target for an attacker
  - Effective attack surface value: Product of the {# of attack surfaces within a root attack vector} and the {attack bias}

https://www.microsoft.com/windowsserver2003/docs/AdvSec.pdf

RASQ Sample root vectors & bias values

Root vector	Bias value	Comment
Open sockets	1.0	Every open & listening socket is a potential target
Open RPC endpoints	0.9	Like sockets but require more skill
Enabled accounts	0.7	Default accounts simplify brute-force password attacks
Enabled accounts in the Administrator group	0.9	Admin accounts are higher risk
Weak ACLs in file system	0.2	Most files in the system are targeted after a system is compromised
Weak ACLs on file shares	0.9	Default shares are commonly known and often targeted

https://www.microsoft.com/windowsserver2003/docs/AdvSec.pdf

Summary

- Better OSes would be nice
  - A secure OS will make it *easy* to write security-sensitive programs
- Minimize chances of errors
  - Eliminate unnecessary interactions (files, users, network, devices)
  - User per-process or per-user /tmp
  - Avoid error-prone system calls and libraries
    - Or study the detailed behavior and past exploits
    - Minimize comprehension mistakes
  - Specify the operating environment & all inputs
    - And validate it at runtime
      - PATH, LD_LIBRARY_PATH, user input, …
    - Don't make user input a part of executed commands

App Confinement

Confinement

- We realize that an application may be compromised
  - We want to run applications we may not completely trust
- Not always possible
- Limit an application use a subset of the system's resources
- Make sure a misbehaving application cannot harm the rest of the system

How about access control

- Limit damage via access control
  - E.g., run servers as a low-privilege user
  - Proper read/write/search controls on files … or role-based policies
- ACLs usually do not have permissions for “don’t allow access to anything else”
- We are responsible for changing protections of every file on the system that could be accessed by *other*
  - And hope users don’t change that
  - Or use more complex mandatory access control mechanisms … if available

Not high assurance

Other resources to protect

- CPU time
- Amount of memory used: physical & virtual
- Disk space
- Network identity & access
- We can regulate access to some resources
  - POSIX `setrlimit()`
    - Maximum CPU time that can be used
    - Maximum data size
    - Maximum file that can be created
    - Maximum memory a process can lock
    - Maximum # of open files
    - Maximum # of processes for a user
    - Maximum amount of physical memory used
    - Maximum stack size

Network identity

- Each system has an IP address unique to the network
- Compromised application can exploit address-based access control
  - E.g., log in to remote machines that think you're trusted
- Intrusion detection systems can get confused

Compromised applications

- Some services run as root
- What if an attacker compromises the app and gets root access?
  - Change resource limits
  - Change file permissions (or ignore them!)
  - Change the IP address of the system

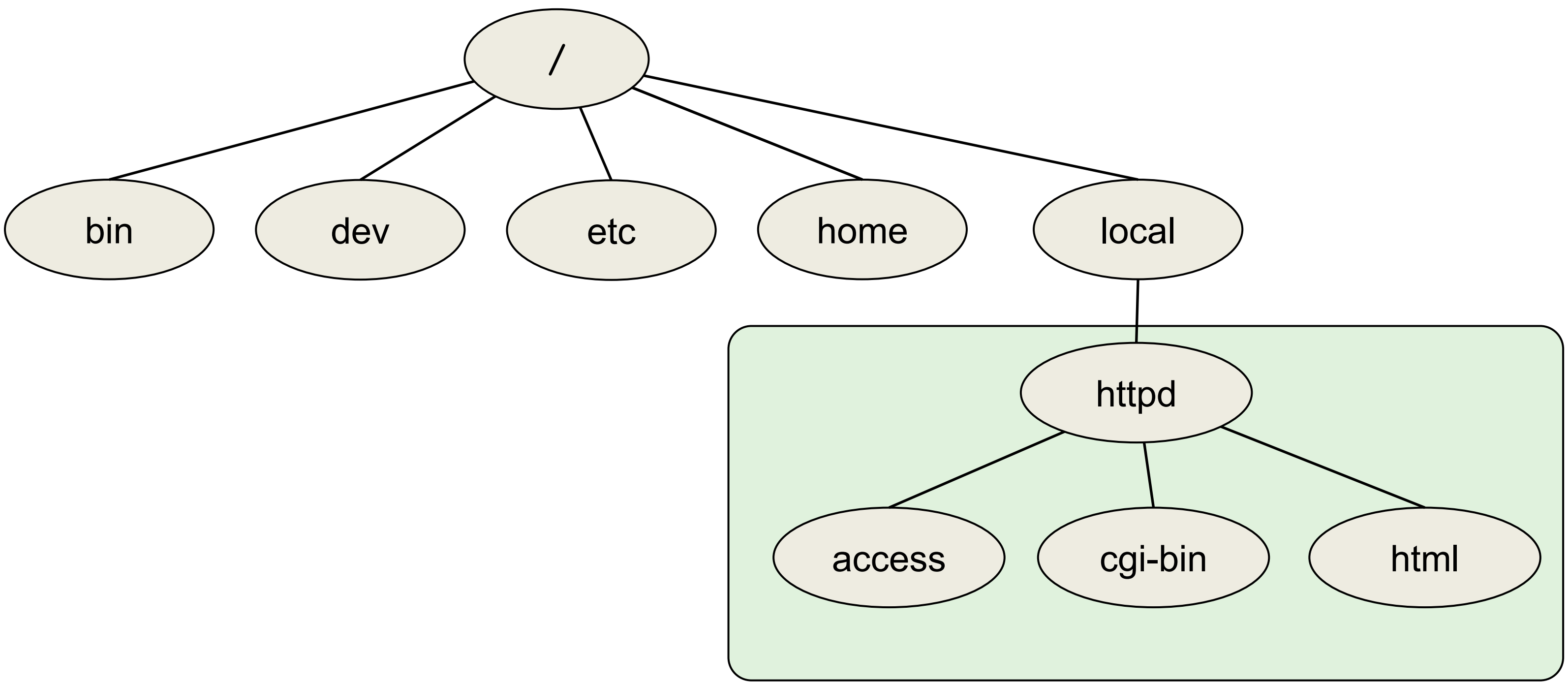
Application confinement goals

- Enforce security – broad access restrictions
- High assurance – know it works
- Simple setup – minimize comprehension errors
- General purpose – works with any (most) applications

We can't get all of this…

chroot: the granddaddy of confinement

- Oldest confinement mechanism
- Make a subtree of the file system the root for a process
- Anything outside of that subtree doesn't exist

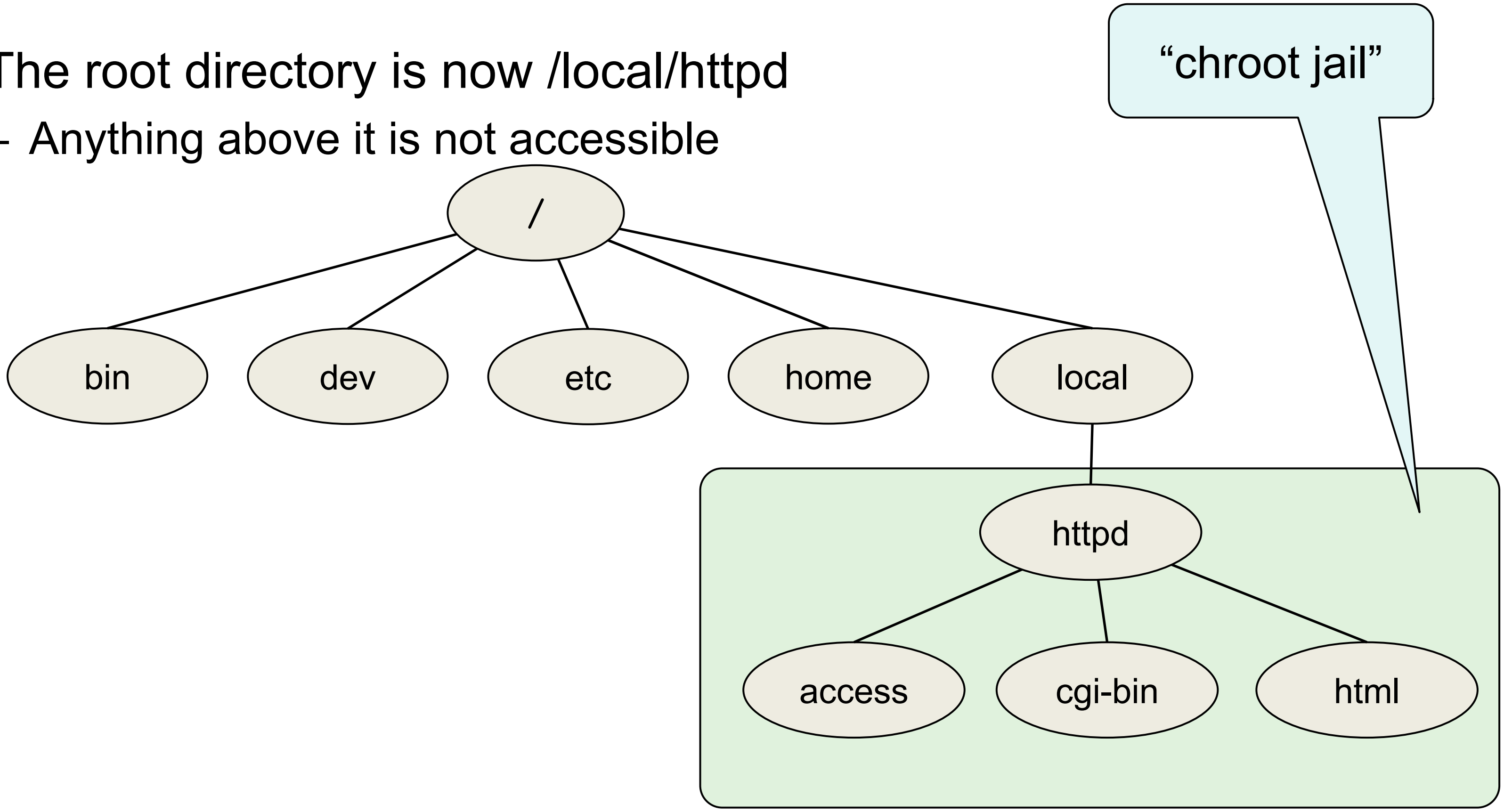


chroot: the granddaddy of confinement

- Only root can run *chroot*

```
chroot /local/httpd
su httpuser
```

- The root directory is now /local/httpd
  - Anything above it is not accessible



Jailkits

- If programs within the jail need any utilities, they won't be visible
  - They're outside the jail
  - Need to be copied
  - Ditto for shared libraries
- Jailkit (https://olivier.sessink.nl/jailkit/)
  - Set of utilities that build a chroot jail
  - Automatically assembles a collection of directories, files, & libraries
  - Place the **bare minimum** set of supporting commands & libraries
    - The fewer executables live in a jail, the less tools an attacker will have to use
  - Contents
    - jk_init: create a jail using a predefined configuration
    - jk_cp: copy files or devices into a jail
    - jk_chrootsh: places a user into a chroot jail upon login
    - jk_lsh: limited shell that allows the execution only of commands in its config file
    - …

https://olivier.sessink.nl/jailkit/

Problems?

- Does not limit network access
- Does not protect network identity
- Applications are still vulnerable to root compromise
- chroot must be available only to root
  - If not…
  - Create a jail directory `mkdir /tmp/jail`
  - Create a link to the su command `ln /bin/su /tmp/jail/su`
  - Copy or link libraries & shell …
  - Create an /etc directory `mkdir /tmp/jail/etc`
  - Create password file(s) with a known password for root
  - Enter the jail `chroot /tmp/jail`
  - `su root` – su will validate against the password file in the jail!

Escaping a chroot jail

- If you can become root in a jail, you have access to all system calls
- Create a device file
  - On Linux/Unix/BSD, all non-network devices have filenames
  - Even memory has a filename (/dev/mem)

1. Create a memory device (*mknod* system call)
   - Change kernel data structures to remove your jail
2. Create a disk device to access your raw disk
   - Mount it within your jail and you have access to the whole file system
   - Get what you want, change the admin password, …
3. Send signals to kill other processes (doesn't escape but causes harm)
4. Reboot the system

chroot summary

- Good confinement
- Imperfect solution
- Useless against root
- Setting up a working environment takes some work (or use jailkit)

FreeBSD Jails

- Enhancement to chroot
- Run via
  - `jail` *jail_path hostname ip_addr command*
- What's different?
  - Can only bind to sockets with a specified IP address and authorized ports
  - Can only communicate with processes inside the jail
  - root power is limited – cannot load kernel modules
  - Hierarchical: create jails within jails
  - Ability to disallow certain system calls
    - Raw sockets
    - Device creation
    - Modifying network configuration
    - Mounting/unmounting file systems
    - set_hostname

https://www.freebsd.org/doc/en/books/arch-handbook/jail.html

FreeBSD Jails: Differences from chroot

- Network restrictions
  - Jail has its own IP address
  - Can only bind to sockets with a specified IP address and authorized ports
- Processes can only communicate with processes inside the jail
  - No visibility into unjailed processes
- Hierarchical: create jails within jails
- Root power is limited
  - Cannot load kernel modules
  - Ability to disallow certain system calls
    - Raw sockets
    - Device creation
    - Modifying network configuration
    - Mounting/unmounting file systems
    - set_hostname

https://www.freebsd.org/doc/en/books/arch-handbook/jail.html

Problems

- Coarse policies
  - All or nothing access to parts of the file system
  - Does not work for apps like a web browser
    - Needs access to files outside the jail (e.g., saving files, uploading attachments)
- Does not prevent malicious apps from
  - Accessing the network & other machines
  - Trying to crash the host OS
- BSD Jails is a BSD-only solution
- Pretty good for running things like DNS and web servers
- Not all that useful for user applications

Linux namespaces

- *chroot* only changed the root of the filesystem namespace
- Linux provides control over the following namespaces:

IPC	System V IPC, POSIX message queues	Objects created in an IPC namespace are visible to all other processes *only* in that namespace
Network	Network devices, stacks, ports	Isolates IP protocol stacks, IP routing tables, firewalls, socket port #s
Mount	Mount points	Mount points can be different in different processes
PID	Process IDs	Different PID namespaces can have the same PID – child cannot see parent processes or other namespaces
User	User & group IDs	Per-namespace user/group IDs. You can be root in a namespace with restricted privileges
UTS	Hostname and NIS domain name	*sethostname* and *setdomainname* affect only the namespace

See namespaces(7)

Linux namespaces

Unlike chroot, unprivileged users can create namespaces

- unshare()
  - System call that dissociates parts of the process execution context
  - Examples
    - Unshare IPC namespace, so it's separate from other processes
    - Unshare PID namespace, so the thread gets its own PID namespace for its children
- clone() – system call to create a child process
  - Like *fork()* but allows you to control what is shared with the parent
    - Open files, root of the file system, current working directory, IPC namespace, network namespace, memory, etc.
- setns() – system call to associate a thread with a namespace
  - A thread can associate itself with an existing namespace in /proc/[pid]/ns

Linux capabilities

How do we restrict what *root* can do in a namespace?

- UNIX systems distinguished *privileged* vs. *unprivileged* processes
  - Privileged = UID 0 = root => kernel bypasses all permission checks
- If we can provide limited elevation of privileges to a process:
  - If a process becomes root, it would be limited in what it could do
  - E.g., no ability to set UID to root, no ability to mount filesystems

N.B.: These *capabilities* have nothing to do with *capability lists*

Linux capabilities

- Linux divides privileges into 38 distinct controls, including:
  - CAP_CHOWN: make arbitrary changes to file UIDs and GIDs
  - CAP_DAC_OVERRIDE: bypass read/write/execute checks
  - CAP_KILL: bypass permission checks for sending signals
  - CAP_NET_ADMIN: network management operations
  - CAP_NET_RAW: allow RAW sockets
  - CAP_SETUID: arbitrary manipulation of process UIDs
  - CAP_SYS_CHROOT: enable chroot
- These are per-thread attributes
  - Can be set via the *prctl* system call

Linux Control Groups (cgroups)

- Limit the amount of resources a process tree can use
  - CPU, memory, block device I/O, network
    - E.g., a process tree can use at most 25% of the CPU
    - Limit # of processes within a group
  - Interface = cgroup file system: /sys/fs/cgroup
- Namespaces + cgroups = lightweight process virtualization
  - Process gets the illusion that it is running on its own Linux system, isolated from other processes

Vulnerabilities

- Bugs have been found
  - User namespace: unprivileged user was able to get full privileges
- But comprehension is a bigger problem
  - Namespaces do not prohibit a process from making privileged system calls
    - They control resources that those calls can manage
    - The system will see only the resources that belong to that namespace
  - User namespaces grant non-root users increased access to system capabilities
    - Design concept: instead of dropping privileges from root, provide limited elevation to non-root users
  - If a real root process has its admin capability removed
    - If it creates a user namespace, the capability is restored to the root user in that namespace – although limited in function

Next time:
Containers & Virtual Machines

The end