Security In-Depth for Linux Software

Preventing and Mitigating Security Bugs

Julien Tinnes   Chris Evans

Google Inc.

October 2009 / HITB Malaysia

Goals of this Talk

1. How to implement *security in depth* and the *least privilege* principle in your Linux code
2. Explain designs of sandboxing techniques on Linux
3. Good code writing and design practices can work

What is Security in Depth?

A secure application should have tolerance for mistakes

- A single failure should not completely break the security model

- Today, we will try to address this from a Linux application programmer perspective
- Using Chromium and vsftpd as examples

Steps to Security in Depth

1. Secure code: reduce number of mistakes
2. Application-level exploitation mitigation (SSP, relro. . .)
3. System-level exploit mitigation (ASLR, NX)
4. Privilege dropping (Sandboxing)
5. Mandatory access control
6. Update strategy

Steps to Security in Depth

1. Secure code: reduce number of mistakes
2. Application-level exploitation mitigation (SSP, relro. . .)
3. System-level exploit mitigation (ASLR, NX)
4. Privilege dropping (Sandboxing)
5. Mandatory access control
6. Update strategy

Outline

1. Privileges in Linux
   - Process and Privileges
   - Privilege-related Facilities
2. Writing Good Code
   - Preventing Common Security Flaws
   - Privilege Separation
   - Trust Relationships
   - Update Strategy
3. Sandbox designs
   - Sandboxing Definition
   - ptrace(), setuid and SECCOMP sandboxes
   - Other approaches
   - Attack surface evaluation

Outline

The Privilege Model of Unix

In a Nutshell...

- Each process has its own address space
- MMU enforces separation of address spaces

- The kernel is a mandatory interface to the system
- The process is the privilege boundary

- `root` has access to everything
- other users are subject to discretionary access control

Privileges Ordering in the General Case

Definition

Process *A* has more privileges than process *B*
if *A* has access to every resource *B* has access to

- Any process running as `root` is more privileged than any other process
- Two processes with the same `uid` and `gid` may have the same privilege
- One can generally not compare two processes with different `uid`s

Processes and Privilege Separation

Threads

- There is no possible privilege separation inside a process (in the general case)
- Exception: NaCl, SECCOMP sandbox

Debugging

If *A* can `ptrace()` *B*, then *A* is more privileged than *B*

Outline

1. Privileges in Linux
   - Process and Privileges
   - Privilege-related Facilities
2. Writing Good Code
   - Preventing Common Security Flaws
   - Privilege Separation
   - Trust Relationships
   - Update Strategy
3. Sandbox designs
   - Sandboxing Definition
   - ptrace(), setuid and SECCOMP sandboxes
   - Other approaches
   - Attack surface evaluation

Standard Linux Process Privileges

Users and groups

- `uid, euid, suid, fsuid`
- `gid, egid, sgid, fsgid` and supplementary groups

POSIX.1e capabilities

- Designed as a way to split `root` privileges
- Introduced in Linux 2.2

uid, effective uid, saved uid and filesystem uid

Definition (Confused Deputy)

A computer program that is innocently fooled to use its ambient authority

- Partial UID switching is mostly useful to avoid confused deputy problems
- It's useless in case of arbitrary code execution, where the attacker has full control of the application
- Only root can use this facility

Linux Capabilities

- Linux divides root privileges into distinct units

Examples

- `CAP_NET_RAW`: Permit use of RAW and PACKET sockets
- `CAP_SYS_ADMIN`: Administrative operations (`mount()`, `sethostname()`, etc...)
- `CAP_NET_BIND_SERVICE`: Binding to reserved ports (< 1024)

Capabilities Limitation

Common Mistakes

1. Forgetting to switch from uid 0
2. A lot of capabilities are root equivalent
   - Useful for *confused deputy* problems

Root only

- Capabilities are a root privilege dropping facility
- Useless to *further* restrict a normal user's privileges
  - Normal users can do a lot

Changing Root

Using *chroot()*

- A popular way to drop filesystem access
  How else do you drop access to o+r files?
- Only available to root

Requires dropping privileges afterwards, or easy to escape:

- Popular re-chroot() technique
- Inject modules, `ptrace()` non chroot-ed process, etc. . .
- Look at capabilities for inspirations

Changing Root

Using *chroot()*

- A popular way to drop filesystem access
  How else do you drop access to o+r files?
- Only available to root

Requires dropping privileges afterwards, or easy to escape:

- Popular re-chroot() technique
- Inject modules, `ptrace()` non chroot-ed process, etc...
- Look at capabilities for inspirations

New namespaces: CLONE_NEW*

Courtesy of Linux Containers (LXC)

Recent kernels introduced new `clone()/unshare()` flags

- CLONE_NEWPID: new pid namespace (2.6.24)
- CLONE_NEWNET: new network namespace (2.6.26)
- CLONE_NEWIPC, CLONE_NEWUTS, CLONE_NEWNS (2.6.19)

Interesting ways to drop privileges, but only accessible by root

Resource Limits
rlimits

Resource limits can be used for security

- `RLIMIT_NOFILE`: can't get new file descriptors. But can still `rename()` and `unlink()`
- `RLIMIT_NPROC`: can't create new processes

- If used for security, soft and hard limit need to be set to zero
- Or attacker could replace an existing fd to create new sockets/access new files

Dumpable (Debuggable) Process

Linux supports a per process dumpable flag

- Can be set through `prctl` with `PR_SET_DUMPABLE`
- Or when executing a file you don't own and can't read
- Or when switching uid

- A process without CAP_SYS_PTRACE cannot `ptrace` a non dumpable process
- Therefore it's an elevation of privileges
- But it allows to lower *another process'* privileges

Mandatory Access Control (MAC)

Linux has several MAC options

- In the Kernel, LSM-based: SELinux, SMACK, TOMOYO
- Outside: GRSecurity, RSBAC, AppArmor (not for long?)...

- Offers some flexibility and lots of options
- But, they require the administrator to set-up a policy

Conclusion on Privilege-related Facilities

- Most of them are designed to give less privileges to root
- Those which don't still require root
- Easy to protect against *confused deputy problems* but not against *arbitrary code execution*

Outline

Mostly a solved problem...

General principle

- Use APIs that are harder to abuse than use correctly

- Strings: use a C++-like buffer encapsulation (even in C)
- Auth: tiny API, all code in one place
- Must be readable

Easy to Abuse API: OpenSSL

OpenSSL API modeled after UNIX API

```
int SSL_read(SSL *ssl, void *buf, int num);
```

What does it mean if that returned "0" ?

Hard to Read Code

```
for (p = old_prompt,
     len = strlen(old_prompt);
     *p; p++) {
 if (p[0] =='%') {
  switch (p[1]) {
    case 'h':
      p++;
      len += strlen(user_shost) - 2;
      subst = 1;
      break;
```

```
new_prompt = (char *) emalloc(++len);
endp = new_prompt + len;
for (p = old_prompt,
     np = new_prompt; *p; p++) {
  if (p[0] =='%') {
    switch (p[1]) {
      case 'h':
        p++;
        n = strlcpy(np, user_shost,
                    np - endp);
        if (n >= np - endp)
          goto oflow;
        np += n;
        continue;
```

Outline

The use of Multiple Processes

- Use one process per "privilege level"
- Use different UIDs
- Each process should run with the minimum privilege it needs
- Have a simple message protocol and transport between processes

Vsftpd

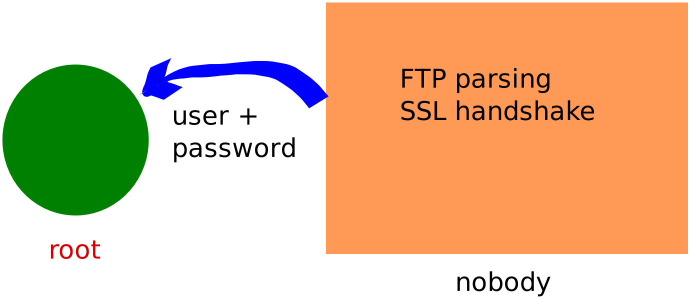
- Pre-vsftpd: anonymous $\Rightarrow$ root

vsftpd scenario

- No anonymous access
- Logins to real accounts over SSL

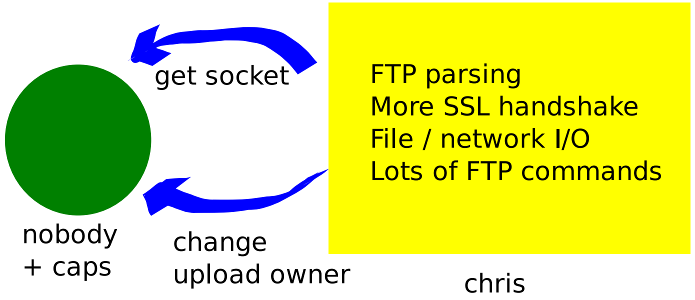
vsftpd: pre-authentication

vsftpd: unauthenticated



vsftpd: post-authentication

vsftpd: authenticated

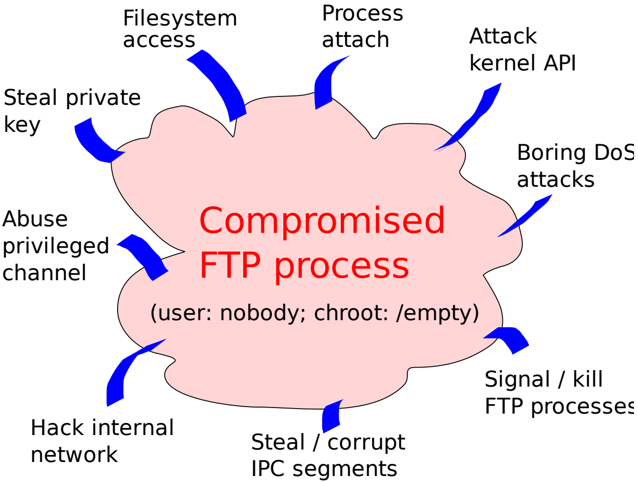


Outline

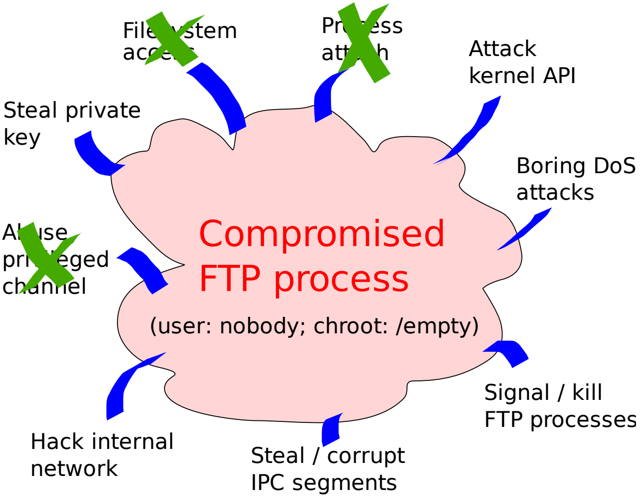
1. Privileges in Linux
   - Process and Privileges
   - Privilege-related Facilities
2. Writing Good Code
   - Preventing Common Security Flaws
   - Privilege Separation
   - Trust Relationships
   - Update Strategy
3. Sandbox designs
   - Sandboxing Definition
   - ptrace(), setuid and SECCOMP sandboxes
   - Other approaches
   - Attack surface evaluation

The Messages Between Multiple Processes

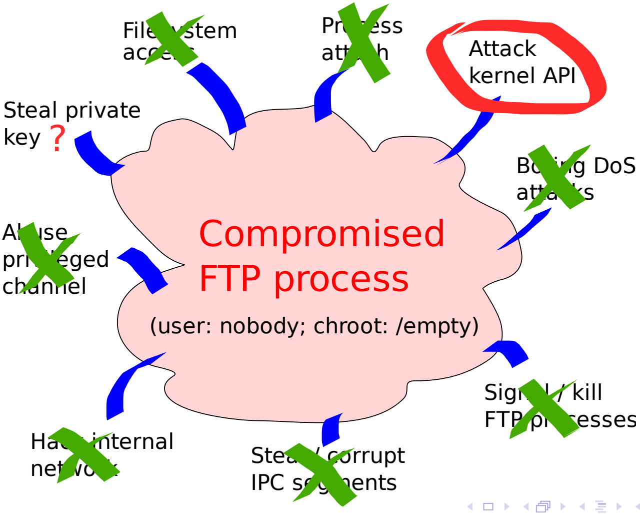
- A higher privileged process must *distrust* requests from a lower privileged process
- Bad messages could simply be garbled
- Or bad messages could be syntactically valid but claim evil things


Filesystem access
Process attach
Attack kernel API
Steal private key
Boring DoS attacks
Abuse privileged channel
Compromised FTP process
(user: nobody; chroot: /empty)
Signal / kill FTP processes
Hack internal network
Steal / corrupt IPC segments

vsftpd v2.0


Filesystem access
Process attach
Attack kernel API
Steal private key
Boring DoS attacks
Abuse privileged channel
Compromised FTP process
(user: nobody; chroot: /empty)
Signal / kill FTP processes
Hack internal network
Steal / corrupt IPC segments

vsftpd v2.2 (default)



More Subtle Trust Examples From Chromium and vsftpd

Chromium

- Uploading local filesystem files to a web site
- Causing memory corruption in the privileged browser via audio-related integer overflows
- Renderer crash and extracting a stack trace

vsftpd

- Sleeping after failed login

Outline

1. Privileges in Linux
   - Process and Privileges
   - Privilege-related Facilities
2. Writing Good Code
   - Preventing Common Security Flaws
   - Privilege Separation
   - Trust Relationships
   - Update Strategy
3. Sandbox designs
   - Sandboxing Definition
   - ptrace(), setuid and SECCOMP sandboxes
   - Other approaches
   - Attack surface evaluation

Secure software and patching

Remember!

Any large piece of software will have security bugs

- Secure design is an important vulnerability mitigation
- Getting fixes to users fast is often overlooked

Outline

1. Privileges in Linux
   - Process and Privileges
   - Privilege-related Facilities
2. Writing Good Code
   - Preventing Common Security Flaws
   - Privilege Separation
   - Trust Relationships
   - Update Strategy
3. Sandbox designs
   - Sandboxing Definition
   - ptrace(), setuid and SECCOMP sandboxes
   - Other approaches
   - Attack surface evaluation

Sandboxing

Sandboxing (in this talk)

The ability to restrict a process' privileges:

- Programmatically
- Without *administrative authority* on the machine
- *Discretionary privilege dropping*

Administrative Authority

- Being in charge of administrating the machine (or Linux distribution)
- One still can do sandboxing as a root process

Mandatory Access Control vs. Sandboxing

Mandatory Access Control

- For administrators and distribution maintainers
- One policy to rule over many programs
- Without the need for control over the code

Sandboxing

- For software developers
- One code that works on many machines
- Without the need to administer the machines

Threat Model of Sandboxing

Here, we assume *arbitrary code execution* inside the sandboxed process

- The attacker fully controls the sandboxed process
- Dropping privileges is useless if it's revertible

We only care marginally about *confused deputy* problems

Outline

Sandbox Designs

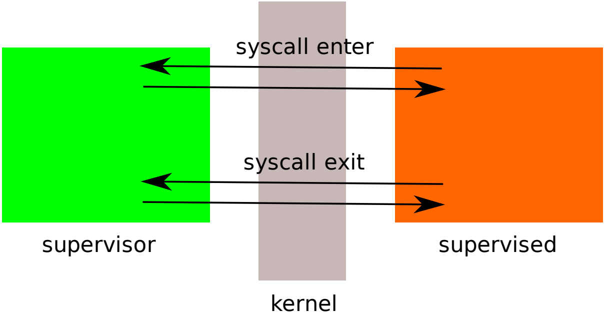
- There are very few facilities to write sandboxes in the kernel
- Most of the one we've presented are only available to root
- Adding new facilities to the kernel is not a short term option

We will present three designs, used in vsftpd and Chromium

- `ptrace()` sandbox (vsftpd experiment)
- setuid sandbox
- SECCOMP sandbox

`ptrace()` Sandboxing

ptrace() sandboxing



`ptrace()` Sandboxing: pros

- Tightly restricts kernel API, lowers attack surface
- High granularity of access control possible
- Can be used securely, despite widely-cited race conditions
- Code relatively simple (but not trivial)

`ptrace()` Sandboxing: cons

- *Very* buggy area of kernel
- Lots of pitfalls
- Performance degradation
- Highly sensitive to exact kernel and glibc version and architecture

`ptrace()` Sandboxing: pitfalls

- Race conditions: don't allow threads (or shared memory!)
- Or don't gate access control on pointer-based arguments
- SIGKILL vs. the supervisor or the supervisee
- 64-bit vs. 32-bit syscalls
- Desynchronizing the supervisor
- Probably best avoided

Setuid Sandbox
(Julien Tinnes, Tavis Ormandy)

root seemed hard to avoid

- Need to drop access to the filesystem
- RLIMIT_NOFILE is not enough (`unlink()`, `rename()`)
- Preventing `ptrace()` on other processes
- Prevent sending signals to other processes

- Switching uid and gid would mostly solve this
- We designed a setuid sandbox

Google

Setuid Sandbox
(Julien Tinnes, Tavis Ormandy)

root seemed hard to avoid

- Need to drop access to the filesystem
- RLIMIT_NOFILE is not enough (`unlink()`, `rename()`)
- Preventing `ptrace()` on other processes
- Prevent sending signals to other processes

- Switching uid and gid would mostly solve this
- We designed a setuid sandbox

Setuid Sandbox

UID switching

- We require an administratively defined pool of UIDs/GIDs
- No need for /etc/passwd entries
- On invocation, search for unused UID/GID
- Switch to them
- Execute program to sandbox

Setuid Sandbox

How to do this statelessly ?

- Choose random UID/GID in the pool
- Use RLIMIT_NPROC to make `setuid()` fail if uid is already used
- If it fails, repeat until pool is exhausted

Preventing a user from exhausting the pool

- Ideal: Partition the pool among UIDs
- Trade-off: Partition the pool against hashes of UIDs

Setuid Sandbox

How to do this statelessly ?

- Choose random UID/GID in the pool
- Use RLIMIT_NPROC to make `setuid()` fail if uid is already used
- If it fails, repeat until pool is exhausted

Preventing a user from exhausting the pool

- Ideal: Partition the pool among UIDs
- Trade-off: Partition the pool against hashes of UIDs

The Need for chroot()

Uid switching leaves a lot exposed

- /tmp races exploitation
- setuid binary execution
  (also matters for kernel vulnerabilities exploitation)

Could we also get chroot()-ed ?

A Setuid Sandbox, chroot() and execve()

Problem: how do I execve() after I chroot?

1. chroot() to an empty directory
2. drop privileges (switch uid/gid)
3. execve() target

No go

A Setuid Sandbox, chroot() and execve()

Problem: how do I execve() after I chroot?

1. chroot() to an empty directory
2. drop privileges (switch uid/gid)
3. execve() target

No go

Solving the chroot() Problem

Naive

- Give CAP_SYS_CHROOT
- That's giving instant root to anyone

Realistic

- Don't go through execve, drop privileges and `mmap()` code
- Not convenient. And dangerous (hello pulseaudio)

Optimistic

- Let's give a process the privilege to chroot() to an empty directory
- Can we do that?

Giving a Process the Ability to Change Root

Sharing the process' FS structure

- Our sandbox (process *A*) spawns a new process *B*
- We use clone, with CLONE_FS so that *A* and *B* share their root directory, CWD, etc...
- *A* drop privileges, *B* waits for a special message from *A*
- When *A* wants to `chroot()`, it send a message
- *B* chroot() to an empty directory, which also affects *A*

CLONE_FS Security Implications

A root process *B* shares its FS with untrusted process *A*

- That's very scary
- Our *deputy* is under untrusted process influence
- Drugged deputy problem ?

Mitigations (in case something goes wrong)

- *B* can drop capabilities (but CAP_SYS_CHROOT)
- And set RLIMIT_NOFILE to 0,0
- Dropping capabilities is mostly useful
  to make RLIMIT_NOFILE effective

CLONE_FS Security Implications

A root process *B* shares its FS with untrusted process *A*

- That's very scary
- Our *deputy* is under untrusted process influence
- Drugged deputy problem ?

Mitigations (in case something goes wrong)

- *B* can drop capabilities (but CAP_SYS_CHROOT)
- And set RLIMIT_NOFILE to 0,0
- Dropping capabilities is mostly useful
  to make RLIMIT_NOFILE effective

Now that we Can Drop Filesystem Access. . .

Can we drop the need for the UID/GID pool range?

Not changing UID and switching to a single, common GID

- Would prevent ptrace() from a sandboxed process to another process
- PR_SET_DUMPABLE to prevent ptrace() among sandboxed process
- What about signals?

Now that we Can Drop Filesystem Access. . .

Can we drop the need for the UID/GID pool range?

Using a new PID namespace (CLONE_NEWPID) (2.6.24)

- Solves many problems
- Open question: how secure is it?

Dropping Network Access

We can use RLIMIT_NOFILE

- What if we require new descriptors (for files)?
- We can share our file descriptors (CLONE_FILES) with a broker process

Using CLONE_NEWNET (2.6.24+)

Can be used to cut access to the network completely

Dropping Network Access

We can use RLIMIT_NOFILE

- What if we require new descriptors (for files)?
- We can share our file descriptors (CLONE_FILES) with a broker process

Using CLONE_NEWNET (2.6.24+)

Can be used to cut access to the network completely

Setuid Sandbox: Conclusion

- Chromium has been adapted to work with this sandbox (the renderer is sandboxed)
- We have a fully-featured version and a Chromium-dedicated version
- Chromium's version uses the CLONE_FS trick and CLONE_NEWPID
- The setuid sandbox is the first-level sandbox in Chromium

SECCOMP sandbox
(Markus Gutschke, Adam Langley)

Secure Computing mode

- Has been introduced in Linux 2.6.10
- A thread under SECCOMP can use limited system calls
  - read()
  - write()
  - exit()
  - sigreturn()

Google

SECCOMP's limitation

Design

- Seccomp was designed with pure computing in mind
- The "4 system calls allowed" design is simple

Too limited for a browser renderer

- No memory allocations (`mmap()`, `brk()`)
- No ability to get new file descriptors (`recvmsg()`)

SECCOMP sandbox design

Trusted thread (TT)

- For each thread under seccomp, we have a trusted helper thread
- UT asks TT to perform system calls on its behalf
- TT validates and eventually performs them
- Even memory allocations will work

Trusted/untrusted code sharing AS ?

- The trusted code needs to be in RX only memory
- The trusted code can't access any volatile memory

SECCOMP sandbox design

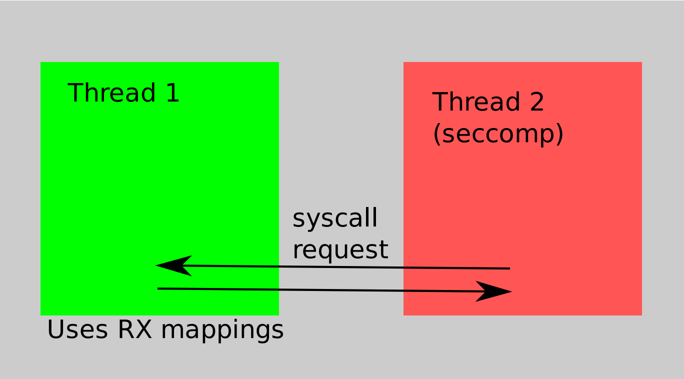
Trusted thread (TT)

- For each thread under seccomp, we have a trusted helper thread
- UT asks TT to perform system calls on its behalf
- TT validates and eventually performs them
- Even memory allocations will work

Trusted/untrusted code sharing AS ?

- The trusted code needs to be in RX only memory
- The trusted code can't access any volatile memory

SECCOMP Trusted and Untrusted Threads



SECCOMP sandbox difficulties

No volatile memory constraint

- The code has to be written in pure assembly
- The code can't use a stack

But we *need* volatile memory

- Many system calls pass pointers to memory (`open()`)
- Evaluating complex system calls in pure assembly would be very hard/impossible

SECCOMP sandbox: the trusted process

Something needs access to volatile memory

- Complexities can be handled in a separate trusted process
- The trusted process can use volatile memory
- It shares pages with the trusted thread
- And can write to them (the trusted thread can only read)

SECCOMP sandbox: conclusion

- Has high potential to isolate the kernel
- Still work in progress
- Has still performance issues
- Not yet enabled by default

Outline

1. Privileges in Linux
   - Process and Privileges
   - Privilege-related Facilities
2. Writing Good Code
   - Preventing Common Security Flaws
   - Privilege Separation
   - Trust Relationships
   - Update Strategy
3. Sandbox designs
   - Sandboxing Definition
   - ptrace(), setuid and SECCOMP sandboxes
   - Other approaches
   - Attack surface evaluation

Relying on a MAC

Creating a generic sandbox by relying on a MAC

- Possible if you have some control over the policy
- Example: SELinux Sandbox

Possible to drop privileges during execution ?

- SELinux supports dynamic transitions

Privilege dropping facilities in the Linux kernel

We have to juggle, due to the lack of discretionary privilege dropping facilities

Recent efforts

- LSMSB
- SELinux type boundaries
- ftrace framework ?

Virtualisation

- Lots of people use virtualisation to separate privileges
- By doing that, they are trying to revert to a known problem: physical machines separation. Of course it's not the case.
- It stil offers the advantage over MAC that it doesn't expose the Linux kernel

Outline

Sandboxes attack surfaces

Different sandboxes expose different attack surfaces

- `ptrace()` / ftrace sandbox
- setuid sandbox
- SECCOMP sandbox

Trusted Path Executable

Can TPE protect the kernel?

TPE usually works by limiting loading native code through execve() / PROT_EXEC mmap()

Different paradigm

- With TPE, vulnerabilities in GNU make or CSH become interesting
- Various interpreters can give you enough control without the need for native code execution
  - Recent demo by dpunk using foreign function interface

Conclusion

- Security in depth is important
- Linux has no real sandboxing facilities
- It's difficult, but possible to write sandboxes on current Linux kernels

Worth it for some software

Containing root

Process running as root can be contained

First requirement is to prevent root -> kernel escalation:

- modules injection
- Access to /dev/mem, /dev/kmem
- Raw I/O

Can also have some use outside of Mandatory Access Control

Linux Capabilities Limitations

The need for uid switching

Don't keep uid zero!

Even if you drop capabilities, you generally *need* to change your uid

- For compatibility reasons, capability model coexists with $uid = 0 \Rightarrow all\_capabilities$
- On any execve with uid=0 or euid=0 you will be granted all capabilities
- Or you can create a root setuid executable and run it

Google

Linux Capabilities: securebits

- Starting with Linux 2.6.26 the kernel supports securebits
- Allows to drop the backward compatibility of capabilities with the old model
- SECURE_NOROOT and SECURE_NO_SETUID_FIXUP

You *still* need to drop uid 0

- Attacker might get a shell without securebits
- Attacker can still backdoor a program executed with different privileges

Linux Capabilities: securebits

- Starting with Linux 2.6.26 the kernel supports securebits
- Allows to drop the backward compatibility of capabilities with the old model
- SECURE_NOROOT and SECURE_NO_SETUID_FIXUP

You *still* need to drop uid 0

- Attacker might get a shell without securebits
- Attacker can still backdoor a program executed with different privileges

Linux Capabilities Limitations
Equivalence to root

Root equivalence

Many capabilities are actually equivalent to root

Linux Capabilities Limitations
Equivalence to root

Root equivalence

Many capabilities are actually equivalent to root

CAP_SYS_MODULE, CAP_SYS_RAWIO, CAP_MKNOD

- execute kernel code
- or communicate directly with devices

Linux Capabilities Limitations

Equivalence to root

Root equivalence

Many capabilities are actually equivalent to root

CAP_SYS_PTRACE

- If you can `ptrace()` any process, you can ptrace a process with all capabilities.
- As explained before: if *A* can `ptrace()` *B*, *A* is more privileged than *B*

Linux Capabilities Limitations

Equivalence to root

Root equivalence

Many capabilities are actually equivalent to root

CAP_CHOWN

1. Change ownership of `/etc/passswd`
2. Modify it

Linux Capabilities Limitations
Equivalence to root

Root equivalence

Many capabilities are actually equivalent to root

CAP_CHROOT

1. Create a working chroot environment
2. Backdoor ld.so or libc
3. hardlink a setuid binary inside the chroot environment
4. `chroot`, launch setuid binary

Linux Capabilities: conclusion

Capabilities are still not widely used

- They can avoid *confused deputy* problems
- But are hard to use effectively in case of *arbitrary code execution*
- It's not necessarily trivial to know which ones are full-privileges equivalent

And they are only a root privileges reduction mechanism