CMPSC 497: Security Mechanisms

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Our Goal

• In this course, we want to develop techniques to **prevent** vulnerabilities from being created
  ‣ What’s a vulnerability?
  ‣ How to find them?
OpenSSH

- Secure remote login software
- Client and server architecture

- Client and server establish secure channel using private key stored on server
- Enabling client to login using password without fear of password sniffing
OpenSSH

• Is secure-critical software
  ‣ Runs as root – needs to be login users
  ‣ Stores and uses a private key that if lost could enable machine spoofing
  ‣ Has access to user passwords that may apply to any machine in the domain

• That is OK, OpenSSH is written in C, so I am sure there are no problems
OpenSSH

• Is secure-critical software
  ‣ Runs as root – needs to be login users
  ‣ Stores and uses a private key that if lost could enable machine spoofing
  ‣ Has access to user passwords that may apply to any machine in the domain

• That is OK, OpenSSH is written in C, so I am sure there are no problems
  ‣ That was a joke…
OpenSSH Vulnerabilities

• Circa 2002
  ‣ CVE-2000-0525 – does not properly drop privileges, allowing local users to execute arbitrary commands
  ‣ CVE-2001-0872 – does not properly cleanse critical environment variables, allowing local users to gain root
  ‣ CVE-2001-1029 – does not drop privileges before reading the copyright files, allows local users to read arbitrary files
  ‣ CVE-2002-0059 – releases certain memory more than once ("double free"), allowing remote attackers to execute arbitrary code
  ‣ CVE-2002-0083 – Off-by-one error allows remote malicious servers to gain privileges.
Retroactive Security

- “Penetrate and patch” as flaws are exposed as vulnerabilities
OpenSSH

• After patching enough of these and other vulnerabilities, what is the impact on?
  ‣ Preventing privilege escalation (to root)
  ‣ Protecting program secrets
OpenSSH

• After patching enough of these and other vulnerabilities, what is the impact on?
  ‣ Preventing privilege escalation (to root)
  ‣ Protecting program secrets

• Not sure whether there are other latent flaws that can be exploited (vulnerabilities)?

• Can we make some change to the design to make such exploits much more difficult?
Retroactive Security

- Several codebases have been extended with security mechanisms retroactively
  - X Server, postgres, Apache, OpenSSH, Linux Kernel, browsers, etc.

- With a variety of security mechanisms:
  - Privilege separation, Authentication, Auditing, Authorization, etc.
Privilege Separation

- Isolate parts of a program into separate processes with different sets of privileges
  - Ideally, small amount of code with large amount of privilege
  - Rest of code can run with low privilege

- What parts of code need privileges in OpenSSH?
Privilege Separation

• Isolate parts of a program into separate processes with different sets of privileges
  ‣ Ideally, small amount of code with large amount of privilege
  ‣ Rest of code can run with low privilege

• What parts of code need privileges in OpenSSH?
  ‣ Code that needs access to root privileges
    • to change UID of child process
  ‣ Code that needs access to critical secrets
    • For setting up secure channels and password authentication
OpenSSH Privilege Separation

- How OpenSSH looks after privilege separation
Privilege Separation

• How do we take a monolithic program and create one or more privilege separated components?
Privilege Separation

• How do we take a monolithic program and create one or more privilege separated components?
  ‣ Need to identify functions that need privileges
    • Functions that perform system calls that need root privileges
    • Functions that use the SSH private keys
    • Functions that perform client authentication
  ‣ Make sure that this boundary is correct
  ‣ Separate these functions from rest of program
    • Make a new program out of these functions
  ‣ Enable communication with the unprivileged component
Privilege Separation

• How do we take a monolithic program and create one or more privilege separated components?
  ‣ Need to identify functions that need privileges
    • Functions that perform system calls that need root privileges

• How do we identify a function that needs root privileges?
Privilege Separation

• How do we take a monolithic program and create one or more privilege separated components?
  ‣ Need to identify functions that need privileges
    • Functions that perform system calls that need root privileges

• How do we identify a function that needs root privileges?
  ‣ Can identify system calls from the program code
  ‣ But, may not be able to determine the syscall inputs
Privilege Separation

• How do we take a monolithic program and create one or more privilege separated components?
  ‣ Need to identify functions that need privileges
    • Functions that perform system calls that need root privileges

• How do we identify a function that needs access to an SSH private key?
Privilege Separation

• How do we take a monolithic program and create one or more privilege separated components?
  ‣ Need to identify functions that need privileges
    • Functions that perform system calls that need root privileges

• How do we identify a function that needs access to an SSH private key?
  ‣ Need to identify buffer that stores the key and all its uses
    • Taint tracking
Privilege Separation

• How do we take a monolithic program and create one or more privilege separated components?
  ‣ Need to identify functions that need privileges
    • Functions that perform system calls that need root privileges
  
• How do we identify a function that needs access to passwords to perform client authentication?
  ‣ Similar to use of private key
Privilege Separation

• Complex task for programmers
• Identifying functions is only part of the challenge
• Other tasks
  ‣ Need to enable communication
    • Remote procedure call (RPC) in terms of interface description language (IDL)
    • Marshalling and unmarshalling data
  ‣ Performance impact
    • What if there are many switches to the privilege separated domain?
Privilege Separation

• Complex task for programmers
  ‣ We would like to automate this task (next time)
Authorization

• Suppose your program processes requests from multiple mutually distrusting clients
  ‣ Clients may be users or other programs (apps)

• You need to ensure that the clients cannot steal or modify each others requests
  ‣ Your program may store resources for clients
    ‣ E.g., database, web server, chat server, window manager, media server, etc.

• Generally, you do this by “authorizing” the use of server resources when processing client requests
Authorization

• Check that a subject (e.g., client user or app) is allowed to perform an operation (e.g., read or write) on a resource (e.g., file or program object)
  ‣ E.g., do you have the permission to read /etc/passwd
  ‣ Does Alice have the permission to view the CSE website?
  ‣ Do students have the permission to share project data?
  ‣ Does Prof. Jaeger have the permission to change your grades?

• An Authorization Mechanism enables these questions to be answered
Authorization

- **Subjects** are the active entities that do things
  - E.g., you, Alice, students, Prof. Jaeger

- **Resources** are passive things that things are done to
  - E.g., /etc/passwd, CSE website, project data, grades

- **Operations** are actions that are taken
  - E.g., read, view, share, change
Authorization Mechanism

- Rather than performing authorization in an ad hoc way, you should add an authorization mechanism to your program
  - On each client request, identify the client subject and check whether that client can perform the requested operations on the requested resources
Authorizing Access

<Alice, /etc/passwd, File_Read>
Authorizing Access

- **Resource user**
  - Read passwd file
- **Resource manager**
  - Response
Authorizing Access

Resource user

Operation request \(\downarrow\) \(\rightarrow\) Response

Resource manager

Reference monitor

Allowed? \(\downarrow\) \(\rightarrow\) YES/NO

\langle Alice, /etc/passwd, File_Read \rangle
authorizing access

resource user

operation request → response

resource manager

authorization hooks

reference monitor

allowed? → yes/no

authorization policy
Retrofitting is Hard

• For authorization
  ‣ X11 ~ proposed 2003, upstreamed 2007, changing to date. [Kilpatrick et al., '03]
  ‣ Linux Security Modules ~ 2 years [Wright et al., '02]

Painstaking, manual procedure

At this point, SE-PostgreSQL has taken up a *lot* of community resources, not to mention an enormous and doubtless frustrating amount of *the lead developer’s* time and effort, thus far without a single committed patch, or even a consensus as to what it should (or could) do. Rather than continuing to blunder into the future, I think we need to do a reality check

- http://archives.postgresql.org/message-id/20090718160600.GE5172@fetter.org
One Example

• Linux Security Modules
  ‣ Add authorization mechanism to the Linux kernel
  ‣ About 200 calls to the reference monitor module
  ‣ To protect files, superblocks, IPCs, processes, shared memory objects, semaphores, devices, etc.

• In the mainline Linux kernel since 2004
LSM Hooks

```c
linux/fs/read_write.c:
ssize_t vfs_read(...) {
    ... 
    ret = security_file_permission(file, ...);
    if (!ret) {
        ret = file->f_op->read(file, ...);
    }
    ...
}
```

Security check function

Security sensitive operation
Challenges

• Need to identify the subjects who make requests
  ‣ Called authentication (e.g., passwords, keys, etc.)
  ‣ May simply want to restrict your server to known clients

• Linux knows about processes, so this was easy
Challenges

• Need to determine the **resources** and **operations** your program needs to protect
  › This can be a harder problem
  › There are lots of resources in complex programs
    • E.g., many, many structured types in the Linux kernel
LSM Operations

• For example, control access to file read
  ‣ But several kernel functions enable file read
  • Forgot to include operation for \texttt{do\_read\_writev}

Security check

\begin{verbatim}
linux/fs/read_write.c:
szize_t vfs_read(...) {
  ...
  ret = security_file_permission(file, ...);
  if (!ret) {
    ret = file->f_op->read(file, ...);
  }
  ...
}
\end{verbatim}

\begin{verbatim}
linux/fs/readdir.c:
szize_t vfs_readdir(...) {
  ...
  ret = security_file_permission(file, ...);
  if (!ret) {
    ret = file->f_op->readdir(file, ...);
  }
  ...
}
\end{verbatim}

\begin{verbatim}
linux/fs/read_write.c:
szize_t do_read_write(...) {
  ...
  ret = security_file_permission(file, ...);
  if (!ret) {
    ret = file->f_op->readv(file, ...);
  }
  ...
}
\end{verbatim}

Same security sensitive operation: file read/write
LSM Operations

• Another problem is called “time-of-check-to-time-of-use” (TOCTTOU)
  ‣ Basically, we may check one resource, but the system may be able to use another

• Check “filp” in sys_fcntl (at top)
  ‣ But, use the fd to retrieve the file pointer later in fcntl_getlk and fcntl_setlk (at bottom)

• Adversary can change the file used

/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,  
               unsigned int cmd,  
               unsigned long arg)  
{  
  struct file * filp;  
  ...  
  filp = fget(fd);  
  ...  
  err = security_ops->file_ops  
    ->fcntl(filp, cmd, arg);  
  ...  
  err = do_fcntl(fd, cmd, arg, filp);  
  ...  
}

static long  
do_fcntl(unsigned int fd,  
          unsigned int cmd,  
          unsigned long arg,  
          struct file * filp) {  
  ...  
  switch(cmd){  
    ...  
    case F_SETLK:  
      err = fcntl_setlk(fd, ...);  
    ...  
  }  
  ...  
}

/* from fs/locks.c */
fcntl_getlk(fd, ...) {  
  struct file * filp;  
  ...  
  filp = fget(fd);  
  /* operate on filp */  
  ...  
}

Figure 8: Code path from Linux 2.4.9 containing an exploitable type error.

THREAD-A:
(1) fd1 = open("myfile", O_RDWR);
(2) fd2 = open("target_file", O_RDONLY);
(3) fcntl(fd1, F_SETLK, F_WRLOCK);

KERNEL-A (do_fcntl):
(4) filp = fget(fd1);
(5) security_ops->file_ops->fcntl(fd1);
(6) fcntl_setlk(fd1, cmd)

THREAD-B:
/* this closes fd1, dupes fd2, and assigns it to fd1. */
(7) dup2(fd2, fd1);

KERNEL-A (fcntl_setlk)
/* this filp is for the target file due to (7). */
(8) filp = fget(fd1)
(9) lock file

Figure 9: An example exploit.
Take Away

• Programs may have lots of ad hoc bugs that prevent it from running securely

  ‣ However, there are certain security goals we may want to achieve
    • Focusing on the goals may make the program easier to protect through security mechanisms targeted for those goals

  ‣ Examples of security mechanisms
    • Privilege separation: Isolate code with extra privileges or sensitive resources from rest of the program – call via small API
    • Authorization: Enforce authorization policy to control which resources an individual client may be able to access