IV. OS Specific Vulnerabilities

Executing Code with Too Much Privileges

Purpose and Contents

- The purpose of this lecture is to
  1. presents vulnerabilities due to bad manipulation of application privileges
  2. presents OS-specific (Linux) privilege-assignment mechanisms and related

The “Too Much Privileges” Vulnerability

Description

- let an application run with higher privileges than strictly required
  - extreme case: administrative / system privileges
- a design flaw
  - contradicts the “least privilege” principle
    - * i.e. give an application the minimum level of privileges needed to get its job done
  - also contradicts the “defense-in-depth” principle
- could also be an implementation bug
  - when the application fails to lower its privileges, when designed to do that
- effects
  - gives an attacker more power when the application is exploited
  - e.g. attacker could execute code with the exploited application’s privileges
  - e.g. attacker could access data normally not allowed to

CWE References

- CWE-264: “Permissions, Privileges, and Access Controls”
  - most general
  - related to the management of permissions, privileges, and other security features that are used to perform access control
- CWE-265: “Privilege / Sandbox Issues”
  - improper enforcement of sandbox environments, or the improper handling, assignment, or management of privileges
- CWE-250: “Execution with Unnecessary Privileges”
  - performs an operation at a privilege level that is higher than the minimum level required, which
  - creates new weaknesses or amplifies the consequences of other weaknesses
- CWE-269: “Improper Privilege Management”
  - does not properly assign, modify, track, or check privileges for an actor, creating an unintended sphere of control for that actor
- CWE-271: “Privilege Dropping / Lowering Errors”
  - does not drop privileges before passing control of a resource to an actor that does not have those privileges

Actual relevance

Related vulnerabilities

- use overly strict access control (permissions) on some resources
  - e.g. give only administrator (root) access to some file or directory
  - ⇒ force running applications with high privileges in order to access that resources
  - not right when lower-privilege users (and their applications) need also access those resources
- fail to drop privileges before executing actions for lower-privilege users
  - forget to do this
  - fail to call the correct functions or specify the correct parameters
  - fail to check if the called functions succeeded or not

Identify the Vulnerability

- generally
  - determine if the application can run correctly with non-admin privileges
- code review
  - determine the (precise) required privileges
  - determine if they are correctly configured
- testing techniques
  - dump the application’s privileges
  - e.g. in Windows get the process token and parse it, or use tools like Process Explorer

Redemption Steps

- run the application with least privilege!
- determine and understand which privileges the application needs
- remove the privileges which are not needed
- could be a complex operation
  - especially when the application need to interact with another higher-privileged one

Linux Privileges and Related Vulnerabilities

Processes

- program = executable file
- process = instance of a running program
- each process normally runs with the privileges of the user that process belongs to
  - user identity (UID) is associated to all his processes ⇒ process' real UID
- processes belonging to root (UID = 0) have absolute access to any resource
- there are cases when a process runs with other privileges (e.g. higher) that those of its user
  - i.e. process "runs as" belonging to another user
  - process' effective UID are considered to determine that process' privileges
- special privileges could be obtained through mechanisms like
  - set-user-id (SUID) and set-group-id (SETGID)

Associated User IDs of a Process

- process user identifiers (i.e. UIDs)
  - real UID
  - saved SUID
  - effective UID
- when a process runs a new program, e.g. by calling the execve() system call
  - its real UID remains the same
  - its effective UID remains the same, unless
    - * the SUID is set for the new program ⇒ effective UID gets the UID of the program owner
  - its saved SUID is replaced with the its new effective UID
- normally a process is allowed to switch its effective UID
  - between its real UID and its saved SUID
- processes with effective UID 0 have complete access to the system

Associated Group IDs of a Process

- process group identifiers (i.e. GIDs)
  - real GID
  - saved SGID
  - effective GID
  - supplemental GIDs, i.e. other groups the effective user belongs to
- processes with effective GID 0 normally does NOT have absolute control of the system

Privileged Programs

- Process’ effective permissions are determined by its:
  - effective UID
  - effective GID
  - supplemental GIDs

SUID and SGID Programs

- process' effective UID / SGID = UID / GID of program's owner
- process saved SUID / SGID = effective UID / GID
- allow normal users to access resources of other users
- process
  - starts with elevated privilege
  - can temporarily drop its privileges by switching to its real UID / GID
  - can regain its privileges by switching back to saved SUID / SGID
  - can permanently drop its privileges by setting its effective and saved IDs to its real IDs
- non-root SUID / SGID programs
  - limited to the operations mentioned above
- root SUID / SGID programs
  - most SUID programs are SUID root, i.e. they belong to the root
  - can also change their IDs arbitrarily

Examples of Non-root SUID and SGID Programs

```
$ find /usr/bin -perm /06000 -print0| xargs -0 ls -l
-rwsr-sr-x 1 daemon daemon   51464 Jan 15  2016 /usr/bin/at
-rwxr-sr-x 1 root   tty      14752 Mar  1  2016 /usr/bin/bsd-write
-rwxr-sr-x 1 root   shadow   62336 May 17 02:37 /usr/bin/chage
-rwsr-xr-x 1 root   root     49584 May 17 02:37 /usr/bin/chfn
-rwsr-xr-x 1 root   root     40432 May 17 02:37 /usr/bin/chsh
-rwxr-sr-x 1 root   crontab  36080 Apr  6  2016 /usr/bin/crontab
-rwxr-sr-x 1 root   shadow   22768 May 17 02:37 /usr/bin/expiry
-rwsr-xr-x 1 root   root     75304 May 17 02:37 /usr/bin/gpasswd
-rwxr-sr-x 1 root   mlocate  39520 Nov 18  2014 /usr/bin/mlocate
-rwsr-xr-x 1 root   root     32944 May 17 02:37 /usr/bin/newgidmap
-rwsr-xr-x 1 root   root     39904 May 17 02:37 /usr/bin/newgrp
-rwsr-xr-x 1 root   root     32944 May 17 02:37 /usr/bin/newuidmap
-rwsr-xr-x 1 root   root     54256 May 17 02:37 /usr/bin/passwd
-rwsr-xr-x 1 root   root     23376 Jan 18  2016 /usr/bin/pkexec
 ................
```

Daemons and Their Children

- long-running processes that provide system services
- usually started automatically at boot time
- often run as root to be able to perform privileged operations
- often run other programs to handle required tasks, and their children are usually also started with root privileges
- could temporarily assume other users' identity to perform certain actions in a safer manner
  - as long as the program leaves its saved SUID and real UID set to 0, it can regain its root privileges later
- example: /bin/login program
  - after the user is logged on, login switches all its UIDs to that user's ID

Example of Daemons and Their Children

```
$ ps axlf | grep -i tty
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1000  1435  1386  20   0  14224   944 pipe_w S+   pts/0      0:00
                                                                \_ grep -- color=auto -i
4     0  1105     1  20   0  65832  3464 -      Ss   tty1       0:00 /bin/login --
4  1000  1333  1105  20   0  22604  5200 wait_w S+   tty1       0:00   \_ -bash
```

User and Group ID Functions

The seteuid() Function. Description

- syntax
  - int seteuid(uid_t euid);
- changes the calling process' effective UID
- use to temporarily change its privileges
- a process with its effective UID
  - root: can set its effective UID to any needed value
  - non-root: can only toggle its effective UID between the saved SUID and the real UID

The seteuid() Function. Examples

- user with UID = 1000 runs a SUID program of user UID = 2

```
$ ls -l suid-program
-rwsrwsr-x 1 bin bin 8936 oct 29 12:50 suid-program

$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=2, saved_suid=2

seteuid(33); // nu se poate schimba effective uid
real_uid = 1000, effective_uid=2, saved_suid=2

seteuid(1000);
real_uid = 1000, effective_uid=1000, saved_suid=2

seteuid(2);
real_uid = 1000, effective_uid=2, saved_suid=2
```

The seteuid() Function. Examples (II)

- user with UID = 1000 runs a SUID program of root UID = 0

```
$ ls -l suid-program
-rwsrwsr-x 1 root root 8936 oct 29 12:50 suid-program

$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=0, saved_suid=0

seteuid(33); // se poate schimba effective UID
real_uid = 1000, effective_uid=33, saved_suid=0

seteuid(1000);
real_uid = 1000, effective_uid=1000, saved_suid=0

seteuid(33); // nu se poate schimba effective UID
real_uid = 1000, effective_uid=1000, saved_suid=0

seteuid(0);
real_uid = 1000, effective_uid=0, saved_suid=0
```

The setuid() Function. Description

- syntax
  - int setuid(uid_t uid);
- for root processes
  - change all UIDs, i.e. effective UID, real UID, saved SUID, to the specified UID – used for permanently assuming the role of a user
    - * i.e. dropping privileges
- for non-root processes → differences on different UNIX variants
  - some (Linux, Solaris, OpenBSD) makes it behave like seteuid()
  - others (FreeBSD and NetBSD) makes it work similar like for root processes
  - ⇒ not recommended for permanently dropping privileges for non-root processes

The setuid() Function. Linux Examples

- user with UID = 1000 runs a SUID program of user UID = 2

```
$ ls -l suid-program
-rwsrwsr-x 1 bin bin 8936 oct 29 12:50 suid-program

$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=2, saved_suid=2

setuid(33); // nu se poate schimba
real_uid = 1000, effective_uid=2, saved_suid=2

setuid(1000);
real_uid = 1000, effective_uid=1000, saved_suid=2

setuid(2); // se schimba doar effective UID
real_uid = 1000, effective_uid=2, saved_suid=2
```

The setuid() Function. Linux Examples (II)

- user with UID = 1000 runs a SUID program of root UID = 0

```
$ ls -l suid-program
-rwsrwsr-x 1 root root 8936 oct 29 12:50 suid-program

$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=0, saved_suid=0

setuid(33); // scade permanent privilegiile
real_uid = 33, effective_uid=33, saved_suid=33

setuid(1000); // nu se poate
real_uid = 33, effective_uid=33, saved_suid=33

setuid(0); // nu se poate
real_uid = 33, effective_uid=33, saved_suid=33
```

The setresuid() Function. Description

- syntax
  - int setresuid(uid_t ruid, uid_t euid, uid_t suid);
- used to set explicitly all three UIDs
- if “-1” is given as argument the current value of the corresponding UID is used
- root processes can set them to any desired value
- non-root processes can set any ID to the current value of any of the three current UIDs

The setresuid() Function. Linux Examples

- user with UID = 1000 runs a SUID program of user UID = 2 and permanently switch to UID = 1000

```
$ ls -l suid-program
-rwsrwsr-x 1 bin bin 8936 oct 29 12:50 suid-program

$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=2, saved_suid=2

setresuid(-1, -1, -1);
real_uid = 1000, effective_uid=2, saved_suid=2

setresuid(33, 33, 33); // nu se poate
real_uid = 1000, effective_uid=2, saved_suid=2

setresuid(-1, 1000, -1);
real_uid = 1000, effective_uid=1000, saved_suid=2

setresuid(-1, 2, -1);
real_uid = 1000, effective_uid=2, saved_suid=2
```

The setresuid() Function. Linux Examples (II)

```
setresuid(-1, -1, 1000);
real_uid = 1000, effective_uid=2, saved_suid=1000

setresuid(-1, -1, 2);
real_uid = 1000, effective_uid=2, saved_suid=2

setresuid(-1, 1000, 1000); // schimba la UID = 1000
real_uid = 1000, effective_uid=1000, saved_suid=1000

setresuid(-1, 2, 2); // nu se poate schimba inapoi la UID = 2
real_uid = 1000, effective_uid=1000, saved_suid=1000
```

The setresuid() Function. Linux Examples (III)

- user with UID = 1000 runs a SUID program of root UID = 2 and permanently switch to UID = 2

```
$ ls -l suid-program
-rwsrwsr-x 1 bin bin 8936 oct 29 12:50 suid-program
$ ./suid-program
INITIAL
real_uid = 1000, effective_uid=2, saved_suid=2
setresuid(-1, -1, -1);
real_uid = 1000, effective_uid=2, saved_suid=2
setresuid(33, 33, 33); // nu se poate
real_uid = 1000, effective_uid=2, saved_suid=2
setresuid(-1, 1000, -1);
real_uid = 1000, effective_uid=1000, saved_suid=2
setresuid(-1, 2, -1);
real_uid = 1000, effective_uid=2, saved_suid=2
setresuid(-1, -1, 1000);
real_uid = 1000, effective_uid=2, saved_suid=1000
setresuid(-1, -1, 2);
real_uid = 1000, effective_uid=2, saved_suid=2
```

The setresuid() Function. Linux Examples (IV)

```
setresuid(2, -1, -1); // schimba la UID = 2
real_uid = 2, effective_uid=2, saved_suid=2
setresuid(1000, -1, -1); // nu poate schimba la UID = 1000
real_uid = 2, effective_uid=2, saved_suid=2
```

The setreuid() Function

- syntax
  - int setreuid(uid_t ruid, uid_t euid);
- used to set real and effective UIDs
- if “-1” is given as argument the current value of the corresponding UID is used
- root processes: set them to any desired value
- non-root processes: OS dependent, but typically change
  - real UID to effective UID
  - effective UID to real UID, effective UID or saved SUID
  - saved SUID tried to be updated, if new effective UID different by new real UID
- useful in the following situation
  - a program is managing two UIDs as its real UID and saved SUID, none being of root
  - the program wants to drop one set of privileges
  - setresuid() function not available
  - solution: setreuid(getuid(), getuid());

Group ID Functions

- setegid: toggle effective GID between saved SGID and real GID
- setgid: change effective GID, and possibly also saved SGID and real GID • setresgid: change all GIDs
- setregid: change real and effective GIDs
- setgroups: set supplemental groups (requires effective UID = 0)
- initgroups: a convenient alternative to setgroups (requires effective UID = 0)
- warning
  - effective UID = 0 ⇒ root processes → the group functions have a special behavior
  - effective GID = 0 ⇒ non-root processes

Reckless Use of Privileges

Description

- context
  - programs running with elevated privileges
- mistake
  - performs potentially dangerous actions on behalf of an unprivileged user without first dropping privileges
  - takes no precautions before interacting with the FS
- results
  - expose important system files, i.e. information leakage
  - compromise the system
- advice
  - a SUID/SGID program should drop (temporarily) its privileges
  - when performing an action normally not allowed to the real UID – alternative: check permissions based on real UID
    - * though, better to drop privileges to mitigate effects in case of vulnerability exploitation

Example

- the SUID root program XF86_SVGS (from XFree86)

```
$ id
uid=1000(test) gid=1000(test) groups=1000(test)

$ ls -l /etc/shadow
-rw------- 1 root root .... /etc/shadow

$ cd /usr/X11R6/bin

$ ./XF86_SVGA -config /etc/shadow
Unrecognized option: root:qEXaUxSeQ45la:10171:-1:-1:-1:-1:-1:-1
...
```

- the program reads any file not checking if the real UID has permissions on that file
- the SUID was not needed for reading the config file, but for making display configuration changes

Libraries

- usage of third-party libraries
- shared libraries are often the source of potential vulnerabilities
- user do not have details about the internals of library functions, they only know the API

Dropping Privileges Permanently

Context

- regular code template to drop privileges permanently

```
// higher privileged operation
// set up special socket
setup_socket();

// common idiom for permanently drop privileges
setuid(getuid());

// non-privileged operations
start_procloop();
```

- in some situations, it could be not enough

Group Privileges

- programs with both SUID and SGID
- **mistake 1**: program forgets to drop group privileges, beside the UIDs
- **mistake 2**: wrong order

```
// drop user privileges
setuid(getuid());

// not having anymore user privileges
// could not lead to a correct drop in goroup privileges
setgid(getgid());
```

  - the saved SGID could remain of the privileged group
  - a possible attacker's executed code could perform a setegid(0); OR setregid(-1, 0); to recover group privileges

Group Privileges

- correct way and order

```
// drop group privileges
setgid(getgid());

// drop user privileges
setuid(getuid());
```

Supplemental Group Privileges

- context
  - processes started as privileged users, e.g. daemons
    - * ⇒ starts with the (supplemental) groups of the privileged user
  - assume the role of an unprivileged user, based on input
- mistake
  - application drops its privileges, but
  - let the supplemental groups of the privileged user to the non-privileged user
- incomplete, vulnerable code

```
if (root) {
  setgid(normal_uid);
  setuid(normal_gid);
}
```

Supplemental Group Privileges

- correct privilege dropping code template

```
if (root) {
  setgroups(0, NULL);
  setgid(normal_uid);
  setuid(normal_gid);
}
```

Non-Root Elevated Privileges

- context
  - a SUID program belonging to a non-root user
- running as non-root the **typical code template is OS dependent**

setgid(getgid());
setuid(getuid());

  - both setuid() and setgid() change only the effective IDs, not the saved IDs
  - attackers exploiting a program vulnerability could regain the relinquished privileges
- solution: use the following functions
  - setresgid() / setregid()
  - setresuid() / setreuid()

Mixing Temporary and Permanent Privilege Relinquishment

- specific to applications that
  - switch back and forth between privileged and non-privileged users,
  - eventually dropping privileges at all, if possible
- subtle errors could occur from the setuid() usage
- example

```
#define RAISE_PRIV seteuid(0);
#define DROP_PRIV seteuid(getuid());   // effective UID <- real UID

void main_loop()
{
    uid_t realuid = getuid();

    // do not need privileges
    DROP_PRIV
    do_unprivileged_action();
    ...
```

Mixing Temporary and Permanent Privilege Relinquishment (II)

```
    // get back high privileges
    RAISE_PRIV
    do_privileged_action();
    ....
    // do not need privileges
    DROP_PRIV
    ...
    // drop privileges permanently
    // !!! not root, so saved SUID not changed
    setuid(realuid);
    ...
}
```

- when dropping privileges permanently the process effective UID is not 0, but realuid, so saved SUID remains unchanged (i.e. 0)
  - an attacker could call (from the possibly compromised program) “seteuid(0);” to regain root privileges

Dropping Privileges Temporarily

Using the Wrong Idiom

- dropping privileges permanently ASAP is the safest option for a setuid application
- but use the correct idiom
- seteuid(getuid());
  - good for temporarily dropping
  - bad for permanently dropping
- good idiom for permanently dropping: setuid(getuid());
  - works as intended only for root

Using More Than One Account

- specific to programs needing to use more than one user account
- example: see details at www.unixpapa.com/incnote/setuid.html
- wrong implementation

```
// become user1
seteuid(user1);
process_log1();

// become user2
seteuid(user2);
process_log1();

// become root
seteuid(0);
```

Using More Than One Account (II)

- correct implementation

```
// become user1
seteuid(user1);
process_log1();

// become root
seteuid(0);

// become user2
seteuid(user2);
process_log1();

// become root
seteuid(0);
```

Permanent Dropping of Privileges for Root Processes - Audit

- 1. when dropping privileges effective must be UID = 0
- 2. supplemental groups must be cleared
  - using setgroups with effective UID = 0
- 3. all three GIDs must be dropped to an unprivileged GID
  - right: setgid(getgid());
  - wrong: setegid(getgid());
- 4. all three UIDs must be dropped to an unprivileged UID
  - right: setuid(getuid());
  - wrong: seteuid(getuid());

Permanent Dropping of Privileges for Non-Root Processes

- 1. cannot modify groups with setgroups()
- 2. for dropping GIDs use
  - setresgid(getgid(), getgid(), getgid());
  - older systems: setgid(getgid()); setgid(getgid());
- 3. for dropping UIDs use
  - use setresuid(getuid(), getuid(), getuid());
  - older systems: setuid(getuid()); setuid(getuid());

Temporary Dropping of Privileges

- make sure code drops any relevant group permissions as well as supplemental group permissions
- make sure the code drops group permissions before user permissions
- make sure code restore privileges before attempting to drop privileges again, either temporarily or permanently

Privilege Extensions

Privilege Limitations in Traditional UNIX

- all-or-nothing privilege model in UNIX
- root has unrestricted access
- example
  - ping needs root privileges to create a raw socket
  - if exploited before dropping its privileges the program has access to all system's resources
- any program requiring special privileges essentially puts the entire system's security in danger

Linux File System IDs

- each process also maintains file system UIDs: FSUID and FSGID
- they address a potentially security problem with signals
  - an unprivileged user could send signals (to attack) to a privileged process, which
  - temporarily having dropped its privileges to that of the unprivileged user
- aimed to be used for all file system accesses
- they are kept synchronized with the effective UID/GID
- used by program that temporarily want to access FS with a normal user's privileges
- could be changed with setfsuid() and setfsgid()
- deprecated in current versions of Linux
  - a signal could be sent only to processes having their read UID that of the sending process

Capabilities

- specific to Linux
- defines a set of administrative tasks (capabilities) that can be granted to or restricted from a process running with elevated privileges
- some examples
  - CAP_CHOWN
  - CAP_SETUID/CAP_SETGID
  - CAP_NET_RAW
  - CAP_NET_BIND_SERVICES
- usage example: ping process could be given only the CAP_NET_RAW capability
- could be applied to both processes and files

Bibliography

- “The Art of Software Security Assessments”, chapter 9, “UNIX 1. Privileges and Files”, pp. 459 – 559
- “Setuid Demystified”, https://www.usenix.org/legacy/events/sec02/full_papers/chen/chen.pdf
- “24 Deadly Sins of Software Security”, chapter 16, “Executing Code with Too Much Privilege”.