hw05

March 2017

1. ls:

   (a) I think what this file does is:

      i. It mimics what actual 'ls' does so that the user doesn't know they're not executing normal 'ls'. So, for example, it calls '/bin/ls' on whatever arguments were passed to it, and then it exits with the correct code from whatever the real version of 'ls' returned.

      ii. If you echo $PATH in the terminal, you'll notice that all directories in the PATH are separated by ':'. The 'tr' command replaces these colons with spaces, effectively creating a variable $DIRS that contains paths to all directories in PATH.

   What this does, then, is copies itself into all the directories in the PATH, simultaneously giving all its classes read and execute permissions. Thus, if you run this 'ls' command it'll copy itself everywhere else in the PATH, effectively spreading across your system.

   (b) I would think that if a user executed a potentially malicious file, that file could've created this 'ls' file, placed it in the user's bin, and then made sure the user's bin was on their PATH. So essentially the attacker needs the user to accidentally execute or open a file on their system, whereby this 'ls' file could start propagating.

   (c) One way the script could better hide its tracks would be to potentially modify the output of '/bin/ls' before returning. So, instead of simply calling the real 'ls' and allowing it to print to the terminal, perhaps what the attacker could do would be to capture the output of '/bin/ls', delete any files named 'ls', and then print the remaining contents out to the terminal. In this way, if the user was in their user bin and typed 'ls' they wouldn't see a mysterious file, even though it's there.

   This file could also try to handle any error messages that might arise from calling it. Let's say it tries to copy itself into a file in a directory, but there's already an 'ls' there - this might raise an error, which would alert a user that something wasn't right.

Lastly, similar to above, let's say a user thinks there's a malicious file on their system but can't find it. To find the file the user might use the linux cmds 'find', 'which', or 'locate'. To really cover their tracks, if the attacker had replaced these files, too, in the user's bin, the attacker could edit the outputs of the commands to effectively hide the locations of any suspicious files.

If the attacker were to combine this with the first attack I described above, then a user wouldn't be able to find suspicious files, and even if they went into the correct directory and typed 'ls', they wouldn't see anything suspicious.

(d) If the script was somehow able to give itself root privileges, then instead of placing itself in the user's bin ('/home/user/bin') it could simply place itself in the systems '/bin'. If it could do this, it could replace actual 'ls' with itself - the 'ls' on your system would then be corrupted. If this was the case, there would be no suspicious file that would pop out as odd to a user, making detection of the attack a lot less likely.

2. delete user:

   (a) Key to this problem is the fact that you're allocating an array of 128 bytes for the password then simply copying the user's input into this array without checking how much you're copying. This is the perfect setup for a buffer overflow attack.

   What I could do as a malicious user would be to use this buffer to write code that calls kill() on all processes in the system, and then continue writing past the limits of the buffer, overwriting the return address of the function such that it returns to, and then executes, the malicious code I wrote.

   Of course, this assumes that the server has an executable stack and isn't making use of something like a canary value to protect against buffer overflow. What I might also be able to do is use ROP to simply jump into code that's been linked, such as returning to stdlib.c, which is included in this file. In stdlib, if I know the layout of the functions in the file, I can then use gadgets to effect a similar attack and get the server kill all processes.

3. aslr:

   (a) If the seed is chosen uniformly from a string of 2^{16} bits then, on average, you'll need to check 2^{15} seed values to successfully mount an attack.

   (b) If this value is correct, then it'll take $2^{15} * .006$ seconds = 196.608 seconds.

   (c) Position independent code can be executed at any address. ASLR is going to randomize the position of the stack, heap, and shared libraries, yet, if a program isn't compiled with position independent code, ASLR might not be able to randomize the position of, say, any shared libraries; if the libraries depend on specific addresses and you start changing their positions, they won't work since they're reliant on particular memory addresses.

4. shebang:

   (a) What an attacker might be able to do is write a script that takes advantage of the small race condition that occurs in run session.sh between when it follows the shebang and reads '/bin/bash' as the superuser and when it actually starts executing '/bin/bash run session.sh'. If an attacker was able to, within that short window, delete the actual run session.sh and have another file, which they wrote, named run session.sh, then this might cause '/bin/bash' running as superuser to start executing the new, malicious run session.sh.

   The big thing to take away is that there's a small race condition between when this binary starts '/bin/bash' as superuser and when it starts executing '/bin/bash run session.sh', and if a user can rename a malicious file to 'run session.sh' within this window, they can execute code they've written as superuser.

5. permissions:

   (a) what these permissions say is that:

      - owner: read, write, and execute permissions. notice the 's' where the 'x' should be, this indicates that the file's sticky bit is set s.t. when executed, this file will run as root.
      - group: read and execute
      - world: only read

   (b) You want to execute the original file because of the sticky bit above - if you copy the file and run it later, you'll run it as a normal user, while if you run the original you'll have elevated permissions as root while you run it.

(c) From what I can tell, ACL allows you to chain together permissions, so for example the gserver binary can be in the same group it's currently in, but you can also specify that your group also has read and execute permissions.

In this case the 'server' would still be able to run the file, as would your group, which would mitigate the problem of changing gerver's group and 'server' no longer being able to run it.

(d)
```
/* assuming my group name's 'my_group' */
setfacl -m g:my_group:5
```

(e) i. If you go into '/course/cs166/student/' and check the permissions of a directory with, say: 'getfacl dfarnha1' you'll get something like this:

```
# file: dfarnha1
# owner: agokasla
# group: cs1660ta
# flags: -s-
user::rwx
user:dfarnha1:r-x
group::rwx
mask::rwx
other::---
```

This is explicitly giving me (but no other cs166 students) permissions to execute and read this file. If you were to check the permissions on another student's directory, you'd see something very similar: their folder would have added user permissions for just their username.

ii. If you go into /course/cs166/student/dfarnha1/cryptography/ivy and check the permissions of the ivy executable, you'll see that other users only have executable permissions - they aren't allowed to read or write to this file. This prevents me from simply opening up the file and looking for the key in the binary.

Something else that might be important relates to how the owner compiled the file. For example, let's say the owner compiled with the '-g' flag. This might mean that I can execute the file, and while executing run GDB on it. If I can do this, then I can step through and see the code. Since I can't do this it's clear care was taken when compiling the file to protect against me seeing the code as its running.