Write-UP - CTF (RA)

Friday, February 5, 2016 12:04 PM

Level 1

- Browsing to the */levels* directory and doing a *ls -al* to see the directory listing, we notice a series of different binaries owned by the different user levels of the game. Also notice that their suid bit is set. This is an important part of the wargame, and most wargames of this fashion. We see that the *level01* binary is owned by *level02*, so if we can somehow exploit the binary we can conceivably recover the password for *level02*. We also see that the source code for the level appears to have been provided in *level01.c*.

- View the source code.

- It appears to be a fairly small C program that doesn't take any command line arguments and simply prints the current system time. Let's execute the binary to validate that it does what it says it does.

- Where would the vulnerabilities in this simple 10-line program exist? It doesn't take any command line arguments, so buffer overflow or format string exploits seem to be out. Let's look at what the system() function actually does. If we type man system we see that it actually executes a shell command using /bin/sh.

- Bingo, we've found the vulnerability! Notice that the program executes the date shell command via system("date"). This is equivalent of executing date directly from the command line. The vulnerability lies in the fact that the program does not specify the full path when attempting to execute the date command. Thus the program is going to rely on the $PATH environment variable of the user executing the program in order to determine where to find the date executable. You can always determine the actual binary to be executed when issuing a command without it's full path by using the which command. For example if we type which date, we see that the current date command that is in our path is /bin/date.

- Since they don't specify a full path to the intended binary, we can manipulate our path and create another executable file called date. Then our version of date will be executed and not the intended /bin/date.

- We have write access to /home/level01. So let's create an executable file named date that will print the contents of /home/level02/.password to the screen. To do this issue the following command:

- echo "cat /home/level02/.password" > /home/level01/date

- Verify that the contents of /home/level01/date appear correct, and then change permissions on the file to be executable via this command:

- chmod +x /home/level01/date

- Okay, so our special date program is setup correctly. Now we need to add it to our path so that it will be executed when running the level01 binary. We do this by adjusting our $PATH environment variable to search our home directory first before looking in other places for the date

program. You can either set your path to only search the home directory or concatenate the home directory to the existing path. The key is ensuring that /home/level01/date is the first instance of any date program found. The following command prepends /home/level01 to our path:

- export PATH=/home/level01:$PATH
- You can verify that our date program is now in our path by issuing the which date command again.
- We're now ready to reveal the contents of the password for level02. Execute /levels/level01 and notice how our password (in my case an md5 hash) is displayed instead of the system time.

Consider:

- Files in the directory
- File permissions
- File contents

HINT:
With this in mind do you notice anything that can be exploited?

How to Solve:
In the date file, participants will find that the program execute the "`print open("/home/level02/.password").read()` " which doesn't have a hardcoded path. They can input their own code here to reveal the password in the target file.

Level 2

- Again we browse to the */levels* directory and do a listing to identify the files for our next challenge.
- We see that the files for our next challenge include a binary entitled *level02* and what appears to be a PHP file entitled *level02.php*. Based on the PHP file it seems this will involve some sort web attack, but let's run the binary to see what information we can discover.
- Our hunch is correct and it appears we need to fire up a browser and see what we have in store. But first, let's take a peek at the source for*level0.php.* I simply ran the cat *level02.php* and copied the output into a text editor (below).
- Looking at this code we can see that it will present a web page with some form of login screen. The PHP code also appears to be manipulating some sort of cookie file. Let's browse to the page and see what it actually does. After pointing to the initial URL, we're presented with a basic authorization dialogue. We know this isn't the part of the challenge, but rather a measure to prevent users from accessing the challenge who haven't yet passed level01.
- Using the level02 credentials authorizes us to view the site and sure enough we're presented with an apparent login screen.
- Okay, so this appears to be drawing information from some sort of cookie that identifies who we are and includes our browser information and IP address. Let's take a closer look at the code to see where we might identify a vulnerability that we can exploit to extract the password for *level03*. On line 37 the page states to echo the *$out* variable to the screen and then present us with the "login" section. On lines 15 and 25, we see that*$out* is assigned based on the condition of a cookie being set with an attribute entitled *user_details*.

- If the cookie is not set, line 15 states that *$out* is assigned the string stating that we're a first time user. It then proceeds to assign a random file name to a text file, sets our *user_details* cookie with the same file name, and stores the file in */tmp/level02/*.
- If the cookie is set, line 25 states that *$out* is directly assigned the contents of the file name presented in our *user_details* cookie. Hopefully, you can see the glaring flaw in this logic, but notice that there is no sanitization conducted on the file name in the cookie prior to assigning its contents to*$out*. The application is simply trusting that the file name being requested by our cookie is the appropriate file stored in */tmp/level02/*. This is a bad assumption, because we can manipulate the cookie and request the contents of any file we would like, provided the user the web server is running as has read access to the file.
- So we know where the vulnerability lies, let's see if we can manipulate our cookie data to request the */home/level03/.password* file instead of the random text file. To do this we will need to use a web proxy of some sort. I'll use Burpsuite for this example, but feel free to use any proxy you feel comfortable with.
- We fire up Burpsuite and set our browser to send requests through it and proceed to browse to our page again. Since we've already tested the functionality, notice that our cookie is set.
- Now here is the excellent part of going through a proxy, we can manipulate whatever data we would like before it's sent to the server. So let's change the file name in *user_details* to *../../home/level03/.password* and see what happens. **(Note: we need to insert the ../../ in order to "escape" out of the /tmp/level02 directory. This is a simple directory traversal technique.)**

And voila! Instead of presenting us with the contents of the cookie file, we are given the contents of the password file!

Consider :

- Web application behavior
- File content

HINT:
What did you observe by accessing the web address and by viewing the source files contents?

How to Solve:
Notice the script reads user data. The filename is based on provided cookie content. Participants cant build a request to read the contents of the level 3 password file for them.

Example:

```
$ curl -u level2 --digest \
       --cookie user_details="../../home/level3/.password" \
       http://ctf.stri.pe/level2.php
```

Level 3

- As before we login to the server with *level03* credentials, browse to */levels*, and do a directory listing.

- We can see a *level03* binary with the suid bit set whose owner is *level04* and we also see a *level03.c* file which would appear to be the source code for the level. Let's determine what kind of file *level03* is by running the file command on it.

- One thing that stands out is that *level03* is a 32-bit executable binary (as opposed to a 64-bit). Let's go ahead and take a look at the source code. I simply ran *cat level03.c* and pasted it into my text editor for easier viewing.

- Briefly skimming the code we see several predefined functions that appear to take a pointer to a character string and manipulate it in some fashion. These functions are labeled *to_upper*, *to_lower*, *capitalize*, and *length*. Also notice that a function entitled *run* exists with a comment that it has been deprecated, but has still been left in the source. We'll come back to this. Then we see a function called *truncate_and_call* which appears to copy a truncated portion of a supplied string to another buffer before passing the buffer as an argument to a function pointer. And finally we see our *main* function.

- If we look at *main* more closely we see that it initially defines two variables, an integer labeled *index* and a programmer defined function pointer array labeled *fns* which contains pointers to the previously defined functions *to_upper*, *to_lower*, *capitalize*, and *length* (but not *run* obviously). Then we notice that *main* expects two command line arguments (beyond the name of the binary itself), namely INDEX and STRING. If the number of command line arguments is not correct, we are presented with a message detailing what is expected and what function is represented by the INDEX value. If we pass the correct number of arguments, the INDEX parameter (*argv[1]*) is converted to an integer and assigned to the *index* variable. On line 80 the program attempts to verify that the value for *index* falls within the expected range of 0-3 by comparing it with a static variable NUM_FNS (which has be set to 4). Note that the comparison on line 80 is only checking to see if *index* is greater than or equal to NUM_FNS. If the *index* value passes the check, the function *truncate_and_call* is called with three parameters – the address of the function pointer array (*fns*), our index, and the final command line argument which is presumably a string. At first glance, the main function seems fairly straight forward.

- if you look closer at *truncate_and_call*, you can see that *argv[2]* gets truncated to 63 bytes with a NULL ('\0') byte tacked onto it before being manipulated in any fashion.

- For this exercise, we control the value of *index* and the first 63 bytes of buffer space passed via *argv[2]*, or STRING.

- Let's look back at the *run* function which is supposedly deprecated. At no point in the code does this function actually get called, but notice that if it were called it would take a string parameter and send it directly to the *system()* command. Remember from *level01* that the *system()* command executes a shell command using the string provided as its parameter. Wouldn't it be nice if we could somehow get that function to execute and pass it a simple string such as *"cat /home/level04/.password"*? Is that possible?

- The answer is yes, and it's done the same way that the other functions in the *fns* array are called. Notice that on line 62, *fns* calls a function presumed to be in the array based on the index value it has been given. The code assumes that *index* has been "sanitized" to only be 0-3, however there is a flaw in the parameter checking logic. It's checking to see if the value for *index* is greater than or equal to 4, but it doesn't check for negative integer values. Thus we might be able to pass a negative index value that happens to point to the address of *run* and get it to execute our command passed via the first 63 bytes of our buffer supplied in *argv[2]*.

- Let's start debugging this binary to see if we can find a way to execute the *run* function.

- load the *level03* binary into *gdb,* set our disassembly-flavor (personal preference), and take a closer at the *truncate_and_call* function.

- line 25 (0x080487ba) which calls edx, or more specifically calls the function pointed to by the *index* variable. Let's go ahead and set a breakpoint here:

- Type the following command into *gdb* and then run the program – *set args 1 $(python -c "print '\x41'*100")*. Notice that we hit our break point just before the *to_lower* function is called (as indicated by index value 1).

- If we do a *info reg*, we see that edx (which is about to get called) holds the address of *to_lower*.

- Let's determine the addresses of some other interesting variables and the *run* function itself:

- Also, we can validate the static addresses for all functions pointed to by *fns* as follows (notice the second value listed matches edx)

- Now we know the static address of *run* which is 0x804875b, and we also have the relative addresses of *fns* and *&buf*. For this run of the program *fns* sits at 0xffe2019c and *&buf* sits at 0xffe2012c. This is excellent because *&buf*, of which we control the first 63 bytes, actually resides higher up the stack than *fns*. Remember that when discussing locations on the stack, the higher up the stack we are, the lower the addresses will be. This is good news because that means we can pass a negative *index* value and *fns* will locate that address which is up the stack as opposed to below. So let's calculate exactly how far apart they are by simply subtracting *&buf* from *fns*.

  `0xffe2019c - 0xffe2012c = 0x70 (which is 112 bytes)`

- So our buffer sits 112 bytes away from *fns.* Remember that when we pass the index value, it's converted to an integer data type which is four bytes long. Looking back at the disassembly, line 20 reflects the equivalent of multiplying our index by 4. So we want to pass an index value that when multiplied by four, will point fns to the top of our buffer. Thus some more simple arithmetic:

  `112 ÷ 4 = 28`

- Bingo, if we pass -28 as our index value, the fns function pointer assume the first four bytes of our buffer is an address of another function and effectively call that address, passing *&buf* as its string parameter. Also note that we can point *fns* to any negative index value in our buffer, from -28 to -13, ensuring that our buffer is padded to maintain 4-byte alignment. Okay, so now we know that we have the ability to execute an address of our choosing. The next section outlines several different options for solutions to successfully exploit this binary and grab the next password. Go ahead and complete the current run of the binary in your GDB session.

- There are obviously multiple ways to exploit this.

  1) `/levels/level03 -21 $'cat /home/level04/.password \x5b\x87\x04\x08'`

- Note that the string *cat /home/level04/.password* is 27 bytes long, thus we added a space at the end in order to maintain 4-byte alignment before entering the address of *run*. Also note that the address in our string is sent in little endian format. Since the command string is 28 bytes (7 integers), we subtract 7 from 28 and provide the index of -21 to ensure we hit the address of the 'run' function. And executing this gives us the password.

  2) Command sequence:

```
echo "cat /home/level04/.password" > $'\x5b\x87\x04\x08'

 chmod +x $'\x5b\x87\x04\x08'

PATH=.:$PATH

/levels/level03 -28 $'\x5b\x87\x04\x08'
```

- In this solution we created an executable file with the shell command to display the password. The executable filename is the same as the address of therun function in little endian, thus it serves two purposes. The first purpose is the address to jump to, and the second is the command to execute.

3) Command sequence:

```
echo "cat /home/level04/.password" > exploit #this could be any
filename of your choosing

chmod +x exploit

ln -s exploit $'\x5b\x87\x04\x08'

PATH=.:$PATH

/levels/level03 -28 $'\x5b\x87\x04\x08'
```

- This effectively does the same thing as solution 2, but creates a symbolic link to an executable with our exploit string command. The link is named the address of *run* in little endian.

- For solutions 2 and 3 it is necessary to update your path to include the directory you're working under.

Consider:

- File type
- Source code content

HINT:

There are a couple of important things to keep in mind with respect to this exploit:
1) The index value is what we use to direct us to the *run* function. Thus it must point to the address of *run* (in little endian).

2) The beginning of our buffer is also the beginning of the string that will be passed to the *run* function as the command to be executed. Thus you can either insert the command at the beginning of the buffer, or create an actual executable file, whose file name matches the little endian address of run.

With respect to best security practices in general:

- Never compile deprecated source code into your binaries. If the code is not needed or not used, you should either remove it or comment it out.
- When conducting input validation, ensure that all cases are covered so that the values you pass to other functions are guaranteed to be correct. This sometimes taking the approach of a different mindset while coding and literally thinking, "how would I break this?"