Rob Dallara

Tianyu Wang

Our Project Experience

The first thing we tried to do when we started this project was crack the passwords of at least two users (which are listed on the “photographers” page) on the Geo Photo website. We wanted at least one of those users to have publisher privileges so that we could eventually access that user’s API key (we assume that only users with publisher privileges on the Geo Photo site have API keys). In order to do this, we first realized that the username/password pairs were likely stored somewhere on the website’s database. Since the website scripts to load photographers on the photographer page, to check the username and password, as well as several other scripts all rely on SQL to access that database, we thought about using SQL injection to do our password cracking. We thought about inputting just the right string in an input box (which would be some SQL statement taking advantage of the way SQL works) or using burp proxy to intercept all the incoming packets (and change their inputs to the database in just the right way) in order to perform our SQL injection. For example, we tried inputting ‘ # ‘ into the “User” box to see if that affected the way the SQL ran in such a way as to produce useful results or even log us in as some random user. Unfortunately, the server simply replied that we had input an invalid username/password pair. We hoped that we could get the webpage to dump out all of the table names of the database. From this, we hoped to perform further SQL injections to ultimately obtain all of the usernames and their associated hashes, which are computed using a combination of the username and password. Since the home page of the website brags about how “securely” the hashes are generated, we created a program called “triplehash.cc” which could compute the hash of a given string using that exact same hash function. We thought that once we had all the username-hash pairs, we could easily crack multiple user passwords with the help of our program.

Unfortunately, we couldn’t figure out a way to perform such SQL injections, and so we resorted to online password cracking. We realized from using the burp proxy that whenever a user tries to log in, a POST request is sent to the server, and the server uses this in order to determine whether or not the person trying to login has entered a correct username and corresponding password. We obviously did not want to try one password at a time manually on a given user since this would have taken an exceedingly long time. So we resorted to writing a program called “password_cracker.py” which tries sending username-password pairs to the website automatically through POST requests with different possible passwords in order to try to crack a user’s password. This program (along with some input files) can be found in the “password cracking” folder of this submission. We started trying this program on the username “june”, and we soon realized from looking at the top of the page on his favorite link that the password he might have chosen some variant of “Coffee Prince” as his password. We tried inputting these variants into the program, and we soon found his password: “coffeeprince”. But when we logged in as him and looked at his account page, we didn't see any API key (which we thought would just be given to us on the account page). We also realized from this that he didn't have publisher privileges. So we then tried logging in as "jgray" (and realized that his password was "balloot" based on his personal description on the "photographers" page). Unfortunately, "jgray" didn't have publisher privileges either. Finally, we tried logging in as "vmccoy". Although we tried some passwords based on key words towards the beginning of his favorite link (such as variants of "alpinism", "mountaineering", and "everest"), we realized that he might have picked a random word from John the Ripper's wordlist of most common passwords. We tried all of those words as passwords for him, and ultimately got "valhalla". Unfortunately, he also didn't have publisher privileges.

We also tried sending over mnemonic passwords (with the help of "pemdas.py" from lab 2) of the phrases "attacked at dawn" and "to know your enemy you must become your enemy" to "fabian" since he sent us (and everyone else) an email with these when another group of students managed to crack his USB key during the project. In addition, we also tried sending over mnemonic passwords to "amw" based on the quotes he lists on a link on his home page (http://cs.unc.edu/~amw/quotations.html). We tried millions of these possible mnemonic passwords of different lengths but none of them worked.

We then realized that NONE of the users might have publisher privileges, and that we needed to somehow escalate the privilege of one of the users in order to access his/her API key. We also realized that the API key for a user would most likely not be given directly on the account page (even for a publisher), and that we would probably have to do some cryptanalysis in order to break it.

To prevent the type of password cracking attack we carried out, the server should have locked a user out of his/her account after a small number (say 3) failed login attempts (whether from the webpage or through POST requests). To prevent offline password cracking attacks, the server should have added functionality to it that would protect against SQL injection attacks (such as filtering out certain characters), and it shouldn't have displayed the hash generation function on its index page!

After cracking the three passwords, we tried to find a way to view the source code of different PHP scripts on the website so we could figure out how to break the authentication tokens we intercepted through the burp proxy after logging in as well as break our own team's password. The first thing we did after logging in was select a user on the home page, hit the "Show" button (so that their photos would appear), and click on one of their photos. We then saw the actual source for that photo, and we tried to download it by trying to save the page (we wanted to see if we could view the image offline). When we tried to do this, we ended up downloading the "download.php" file (which was actually an HTML file (download.php.html), and so we could not view the actual PHP code of "download.php"). However, we realized that by changing the "file" parameter, we could view the sources of different files. We then realized that the server was most likely a UNIX server, and thus most likely had a /etc/passwd file inside its root directory containing information about users and their passwords. However, we didn't know how far up the root directory was relative to the directory of the "download.php" file, and so we tried navigating into higher and higher directories until we got to the directory ../../../../../../etc/passwd. There, we found a list of usernames, but the password fields all had x's in them. We realized from this that the actual passwords would be found in the /etc/shadow file, but we couldn't view its contents (most likely because the user we were logged in as didn't have publisher privileges). We also found the ../../../../../../bin/bash file, but it provided no useful information (although it had quite a bit of plaintext in it at some point). In order to figure out which PHP files were generating the tokens as well as dealing with our team's 30 character password, we developed a directory probing program called "dir_php.py" which would check to see if a particular directory on the server had particular files that we were looking for (namely, PHP scripts). Ultimately, we ended up finding the files manually based on the fact that the server was an Apache server and that its files were stored in the document root "/var/www". We used this to navigate to the "index.php" file (which loads the home page of the website) at http://normandy-vm-04.cs.unc.edu/download.php?file=/../../../../../../var/www/index.php. The home page did not display in the way it did when we first navigated to the home page, and we decided to view the page source, which showed the PHP source of that page! (We had originally assumed this would not happen, because a user usually can't see the PHP of a webpage by viewing the page source: he/she can only view the rendered html. This may have occurred because we had logged in before navigating to the "index.php" file). Based on the source code in this file, we quickly realized that there was a "header.php" file that was used before the page was finished loading. After navigating to this page, we figured out how the tokens we had intercepted earlier were generated. We also viewed the "forgotpw.php" file as "account.php", "footer.php", and other files (although the latter file source codes didn't help us very much).

After all of this, we realized that when we typed in our team's username in the "User" field and then clicked on the "Forgot Password?" button on the Geo Photo home page, we first ended up on a web page in which we had to enter a three character CAPTCHA. After doing it correctly, we ended up on a webpage which prompted us to "enter the token below" in order to retrieve our team's password. However, no token was actually displayed on the screen. Nonetheless, we knew that the "forgotpw.php" file took care of all of this, and that it was the file that generated the token. We then decided to look at it in order to figure out how the token was generated. We quickly realized that the token was generated based on the number of seconds since the UNIX epoch, the number of microseconds into the current second divided by 1000, and our username. We realized that the only thing that was uncertain and probabilistic when the token was generated was the time in microseconds divided by 1000. We also realized that if we typed in the wrong token, we would end up back on the CAPTCHA page. We would then have to get the CAPTCHA right again in order to end up on the password obtaining page again.

In order to crack our team's password, we wrote a program called "timebomb.py" which repeatedly sent POST requests to http://normandy-vm 04.cs.unc.edu/forgotpw.php?user=DallaraWang which involved a particular CAPTCHA and its corresponding correct answer. The server would then respond back with its corresponding webpage for the token prompt, and our program would then randomly guess the number of microseconds/1000 as well as the current time in seconds (based on what we had just seen earlier in UNIX). However, we also had to allow for variations in the server time in seconds because we ran the program a few seconds after viewing the current time in seconds in UNIX. In addition, we quickly realized that different servers often have slightly different times since the UNIX epoch. Thus, we had to account for that variation in our program as well. Although the probability of getting the right token was quite small on a given trial, the program ran thousands of times and eventually output the html of the resulting page when we finally got our team's password, which was "QQyiaDakFMD1t73J40PBYWIdfH9WCK" (This password can also be found in the "wuala" file in our project submission…).

To fix this vulnerability, I would have added a feature to the "forgotpw.php" script which generated the tokens with far more randomness. I would have also added a feature which detected large numbers of POST requests from a single source in a short time frame. This feature would stop responding to that source when the number of POST requests from it in a given time frame gets too high.

We also tried to break token 1 from the server. In order to do this, we looked at how it was encrypted, and we realized that it was encrypted using a certain key. This key was generated using a number of different features, from the current server time to the user and group id's for whom the token was generated. We narrowed down certain features such as the server port (which was port 80) and the remote address (which turned out to be the IP address of the computer we were using). We analyzed the /etc/passwd and /etc/group files to try and figure out what user and group ids we were going to need to use. In the end, we realized that the user and group id are both 65534 when the user is not actually logged in to the Geo Photo server. We also figured out all of the other features except for the server port and getmypid() (which seems to be random anyway…). We also had to deal with some uncertainty of the server time due to the fact that it isn't the same for every server. Unfortunately, this search space was too large for us to break token 1 in a reasonable length of time, and so we didn't quite break token 1. Nonetheless, we did write a program called "token1_cracking.py" which tries to break token 1 for a particular case in which the tokens were generated when we didn't actually log in to the server.

To make token 1 more secure, I would have made the key generation a lot more random. For example, I might have also incorporated the number of microseconds into the current second as well as some other things to dramatically increase the entropy of the generated key.

After finishing all of this, we then tried to write an exploit called "photo.jpg" for the Geo Photo application which took advantage of its mode which did not use watermarking. Before stepping through the program, we first generated our shellcode to display the message box. We did this by running Metasploit on the normandy-vm-01.cs.unc.edu virtual machine after logging in. We started Metasploit using the "msfconsole" command. Afterwards, we typed in "use windows/messagebox" in order to specify that we were generating shellcode for creating a pop-up message box. We then typed "info" to figure out what attributes the message box had. We quickly realized that the message box had a "TEXT" attribute for our chosen text. We then typed: set TEXT "DallaraWang: To be rather than to seem". This set the text of the message box to our given choice. Finally, we typed "generate -e x86/call4_dword_xor -b \\x00 -t raw -f myshellcode.bin" in order to generate our shellcode in a manner that ensured it had no null terminators, which would have stopped us from writing our exploit further onto the stack during Geo Photo execution. After all of this, we downloaded the finished shellcode to our remote desktop from inside the command prompt on our remote desktop in the "C:\Program Files\PuTTY" directory using the following command:

```
pscp.exe -P 1000 dallara@normandy-vm-01.cs.unc.edu:myshellcode.bin" "c:\Documents and Settings\DallaraWang\My Documents"
```

We were then prompted to enter our password one last time to finally download the shellcode to our remote desktop. (Note: in the end, our shellcode "TEXT" attribute wasn't changed to our set value when we actually generated it for some reason. This may have been due to a bug in Metasploit).

We then stepped through the Geo Photo application as it ran and figured out where the image processing took place as well as other key areas of the code (such as its main method). We kept looking for a method which would write input to the stack and not check its length (so that we could keep writing more and more data to the stack until we ran off the end of the stack and generated an exception). I realized that the program just seemed to be reading the image: it wasn't writing to the image file in any way while running in the mode which did not use watermarking. Finally, we realized that one particular piece of image metadata ("Exif") was written to the stack (not a pointer to it: the actual data) without any bounds checking. We quickly formed an exploit which was long enough to write past the end of the stack (and thus generate an exception when the program ran). We also added an address to a "POP POP RETN" instruction sequence in just the right place in the exploit so that that address would overwrite the address of the SEH handler on the stack when our input was provided. This ensured that when the stack exception was generated, the code would jump to the address in the SEH handler, which we now specified. The input also contained our shellcode before the address to the "POP POP RETN" sequence as well as a couple of jump instructions to ensure that execution redirected to our shellcode properly. When we finally ran the program with our input image, a message box popped up saying "Hello, from MSF!"

To fix this vulnerability, I would have tried to avoid writing the image metadata itself (including "Exif") to the stack if at all possible. If this wasn't possible, I would have somehow provided bounds checking to prevent a stack smashing attack such as ours. For example, I might have used the strncpy() function (which does provide bounds checking) as opposed to some other function which doesn't provide bounds checking, such as strcpy().

We then stepped through the Geo Photo application as it ran in watermark mode and tried to figure out if there was vulnerability somewhere in the code which would allow us to inject our shellcode if we provided the right input image while running exclusively in watermark mode. We soon realized that the application would insert a "Watermark" into its input image if one didn't exist when running in watermark mode. We also realized that when we ran the Geo Photo application in watermark mode and provided an input image which already had a watermark in it, the program would find the watermark, display it on the screen, and "verify" it (which involved putting the actual watermark on the stack). We realized that the created watermarks in the image were always 16 bytes long. When we first tried to lengthen the watermark in the image, the program only processed the first 16 bytes of that watermark. Initially, we thought that the program had hardcoded that value into itself and would always check for 16-byte watermarks, but we eventually realized that the value of the length of the watermark was stored in the file just before the watermark. When we changed the watermark length value to higher values, the program would display the corresponding number of bytes to the program output, but the program would stop writing to the stack after a certain point. We eventually realized that this was due to a check in the processing of the watermark onto the stack which stopped the writing of the watermark onto the stack after some hard-coded number of bytes was written. Nonetheless, we were able to write just enough bytes onto the stack to overwrite the SEH handler and have a small jump instruction right afterwards which would jump back to the area of the stack just before the SEH handler address. We planned on putting a jump instruction in that area which would jump back to the beginning of our injected watermark and execute our shellcode.

We then tried to generate our shellcode for "pwn.exe" by using Metasploit in much the same way as before. However, we typed in "use windows/exec" instead of "use windows/messagebox" in order to make sure our shellcode would redirect to our given executable. In addition, we used the command "set CMD pwn.exe" instead of the "set TEXT" command because we were specifying the attribute of an "execution redirection" shellcode instead of an attribute of a message box shellcode. When we generated our shellcode, however, it was too big to fit in the small space between the beginning of our watermark and our SEH handler address. We couldn't figure out how to make the shellcode small enough to fit in that small space, and so we weren't quite able to finish writing the exploit for the second part of part 2 of the project. Nonetheless, to fix this vulnerability which I found in this part of the Geo Photo application, I would have always read in 16 bytes when reading in the watermark from the file regardless of how long it actually was, and created error messages for when the watermark was too big or too small.

In conclusion, we managed to break the passwords of three users, view critical files and the source codes of many server PHP files, obtain our team’s password, and finish the first exploit for part 2 of this project. Although we tried valiantly to break token 1 from the server and write the second exploit for the second half of part 2, we weren’t quite able to finish these. The only things we did not get around to trying were attempting to break token 2 and trying to crack the API key of a user on the server. Nonetheless, if we had had more time on this project, we could have probably cracked token 2 and the API key that went along with it using cryptanalysis. We could have also broken token 1 and completed the second exploit for part 2 of this project.