PCTF Report

Team: Black Shadow

2 May 2018

Cephas Armstrong-Mensah

Debarati Bhattacharyya

Avinash Mathad Vijaya Kumar

Albert Patron

Tien Yeu Yang

Artem Zaets

ASU CSE545 – SOFTWARE SECURITY, SPRING 2018

Table of Contents

Introduction

This has been an interesting semester and this final PCTF capped it off nicely. Black Shadow as a team we worked closely together over the weekends and chatting via our slack channel and google group for disseminated emails. Our plan going into the final PCTF was to have two teams, an offense team and a defense team. The offense team was responsible for exploiting vulnerabilities and the defense team was to patch found vulnerabilities, so we were protected during the hour and fifty minutes of awesomeness.

Following in this report will include but not limited to:

- Project Idea
    - Why we chose it
- How we implemented it
- How well the tool(s) helped
- What could have been done better to improve the tool
- Lessons learned

Project Idea

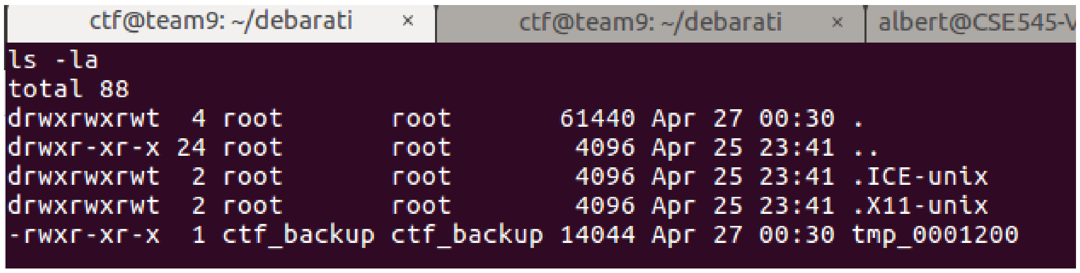
After our last CTF, we determined to be successful in the PCTF we would need a few tools, hopefully automated and based on the different themes of the class – Application Insecurity, Network Insecurity and Web Insecurity. After brainstorming we came up with *backdoor, proxy, reflector, traffic analyzer, vulnerability detector*, *and lastly shellcode*:

- **Backdoor**: Using most of the code from Assignment 1 backdoor application a simple backdoor was created that connects using a raw socket over any desired port and takes in a command, launches this command in a shell, then returns the output. The way this backdoor was implanted was to first exploit an application and get shell access. Once shell access is achieved a public link was created to download the backdoor using wget, public link shared below:

  https://www.dropbox.com/s/xal0baxjt9pt0nf/backdoor?dl=1

  This was coupled with a python script that automatically connected to this backdoor and searched for the current team's flag.

  One interesting note about this backdoor was since xinetd was the network service that called the original service (lets say Backup) this caused an interesting permission dilemma. Since xinetd was running under root it had the ability to run an application under whatever user, in this case it would call the original service under the user "ctf_service name" (i.e. ctf_backup). So, when the malicious user gets shell access in the service it would have the ability to then launch the backdoor under the user "ctf_backup". The reason this causes an interesting permission issue is because the user running under that machine is "ctf" which would not have permission to delete the backdoor, nor kill it's process. The only way to achieve this would be if the user "ctf" were to exploit themselves and delete the process and backdoor under "ctf_backup" username. In Figure 1 is a screenshot showing this, backdoor's name was disguised as a temp file in the /tmp directory:



```
ls -la
total 88
drwxrwxrwt  4 root       root       61440 Apr 27 00:30 .
drwxr-xr-x 24 root       root        4096 Apr 25 23:41 ..
drwxrwxrwt  2 root       root        4096 Apr 25 23:41 .ICE-unix
drwxrwxrwt  2 root       root        4096 Apr 25 23:41 .X11-unix
-rwxr-xr-x  1 ctf_backup ctf_backup 14044 Apr 27 00:30 tmp_0001200
```

Figure 1. Showing renamed backdoor file as tmp_001200

It is understood this was done in this manner to prevent one service exploit from leaking flags from another service, since they would be only readable to the user "ctf_thatOtherService", but this caused a double-edged sword giving an attacker the ability to run a backdoor that was pretty stealthy and somewhat difficult to remedy.

- **Backdoor Auto flag search:** This was created so that once backdoor has been planted on a victim it would be simple to steal their flag. The way this worked was to connect using a socket, ip address was victim teamX and port was whatever port we set when launching the backdoor. Little configuration was required in this script since the team names and IP addresses in the practice PCTF and the real PCTF were the same. Only modifications were the Game IP and our team token. This tool first calls api.py retrieving all of the user's flag ID for the current tick. Then it would attempt to create a socket connection to the teams IP and port. Once this connection was made it would send the command:

  $ ls /CURRENT/PATH/TO/SERVICE/flag_folder/ | grep CURR_FLAG_ID

  Then it would get the entire filename the flag was stored in. then resend another command:

  $ cat /CURRENT/PATH/TO/SERVICE/flag_folder/File_To_Flag

  The return of the previous command would give the current flag for that team. This would cycle through all teams known to have the backdoor installed. This tool ended up being the most effective tool during the final PCTF due to the low amount of configuration and modification required.

- **Proxy**: Binaries used in this CTF were not directly connected to the internet, instead used xinetd to interact over the network with users over sockets. This was done by xinetd interacting directly with the binaries' standard input/output. Since this was the case putting another application in place of the original service to act as a man in the middle between xinetd and the original service was done using Python pwntools. Pwntools allows to fully interact with another application's standard input/output while being able to interact and modify the input and output as we pleased. From here the ability to modify user input and/or the service running output was possible and the creation of a basic firewall, a service to verify outcoming traffic if flags were returned, and a reflecting service to send every other team this same traffic once the communication has ended.
  The idea of the firewall was to have different error levels within this firewall; error level 1, 2, and 3. Error level 1 was intended to only perform sanitization to banned level 1 characters, such as ";", "'". These characters were sanitized simply replacing said characters with the "\" escape character with the original character. Level 2 errors were checked against another list of level 2 banned characters, but instead of trying to sanitize this input the users input was stopped there and returned with a "invalid characters" message then prompting the user to re-enter a value. Lastly level 3 banning characters would shut off the user that was trying to communicate with the application, because these were nasty characters (such as non-ascii characters) that the user was trying to use that are known to have no practical use. There was also a rule created against long inputs that were attempting to crash the original service. The only issue with this proxy was the output, making sure it matched the original services output in the way it handled

newlines and spacings. Having built this after the practice PCTF there was no real way to test against the professor's bots that would send proper input and be expecting proper output. The proxy's job was not only to act as a input sanitizing firewall it was also built to launch a reflector that would reflect this traffic to all other users on the network. Another service that was built was a service to verify the user's input with the actual flags, since the ability to read our own flags was possible. Below is an image visually describing the proxy:

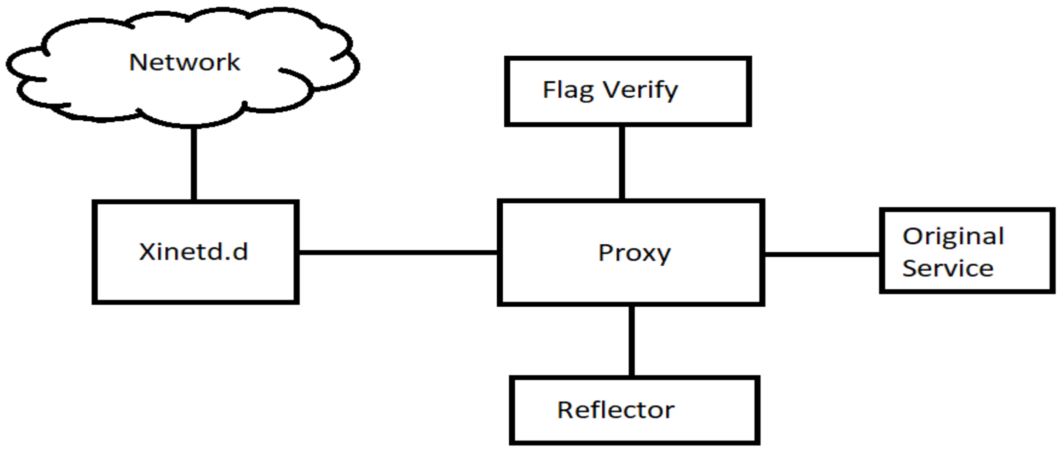


Figure 2. Workflow from Network to Requested Service

- **Reflector**: The proxy would call the reflector that was a different service capable of listening on a port separate from the proxy. Once the user has ended its current session with the proxy it would store the users input in a queue then send it over the network to the reflector. The reflector then uses this, connects to every team sending the previous input sent. The out of this was supposed to be monitored to detect any interesting traffic coming back to the reflector. If so, then the input should be used manually to look further into this as it might be a possible exploit. This service was meant to run on the local machine of the user instead of the virtual machine used to participate in the PCTF. This is due to the high traffic and computation required to be sending this traffic to everyone. The proxy had a static variable that would disable sending its user traffic to the reflector service in case any trouble occurred or unreliability of the reflector.

- **Flag Verify:** Flag verifying was done by reading every flag file when first executed. The files were then parsed through gathering the Flag_Ids, Flag_Passwords, and Flags. This service was to run standalone separate from the proxy in order to avoid reading the flags every time this check was made, causing a lot of overhead on the machine. This service binded to a port and checked user input and output. If a valid flag or flag password was detected in the user's output then the user's input must have either the password or the flag itself, if this output was not detected this service will return 1, otherwise return a 0. This was done instead of just banning all traffic with the tag "FLGxxxxx" because bot traffic could be legitamite where the correct input expected a correct flag since it could not be assumed the bots did not have this information. Also in the event a user was to spam the service with fake flags, which was surprising to see this occured a lot during the PCTF. This service was also coupled and supposed to be called from proxy, and disabling this feature could be done setting a static variable to 0.

- **Port Scanner:** Port scanning was created in an attempt to read any ports that were open on teams other than the standard services that everyone had. This would have revealed things such as hidden backdoors other teams have implanted or any custom services they might have built. Finding this would have been interesting trying to connect to them or trying to make extra services crash by sending malformed traffic.

- **Auto Exploit and flag submission:** A sample tool was built that calls api.py and gathers each team's Flag ID and host name. The tool has a sample dummy exploit that worked on Backup but required to be modified in order to work on any of the services on the PCTF. This tool was built for both running locally and remotely on the same machine. This tool then would send this same exploit to every team using the information gathered from api.py. Finally, it would get all flags gathered and submit them, again using functions from api.py.

- **Traffic analyzer**: The network traffic analysis was an essential part of our toolset. We realized the importance of being able to monitor what is happening on the network and turning this knowledge into advantage was the primary purpose for this tool.

  For this Project CTF we were able to familiarize ourselves with the game environment before the actual competition and had discovered quite a few restrictions that have limited our ability of using any convenient 3rd-party tools on the game server. Since no root-level privileges were granted to participants this time, no custom-made sniffer could be implemented nether in Python nor any other language due to specifics of setting sockets into promiscuous mode for sniffing raw traffic, which in Unix systems requires sudo access. Same problem prevented us from employing Scapy library in our scripts.

  With all these restrictions the only option left was tcpdump, a general-purpose Unix utility for capturing network packets. The problem is, tcpdump provides no option for any sort of analysis for captured packets, meaning no on-air traffic sniffing could be easily implemented. Decision was made to proceed with tcpdump as a way to collect packets of interest, store those into .pcap files and then use python script to process and analyze captured traffic.

  First step was to learn the tcpdump command options and tailor it in a way that any junk traffic would be filtered out and only useful packets collected. It happened to be tcpdump has very powerful command options, including logical expression that allow very precise setup of rules and filters. The resulting command arguments turned out to be the following:

Command argument	**Description**
tcpdump	*Name of utility*
-i eth0	*Specify network interface that connects to the game LAN*
-A	*Display any packet information (IP+TCP headers, payload) in ASCII format*
-q	*Display less protocol information*
-n	*Do not resolve host names (only IP address and port #)*
'tcp[13]=24'	*This mask allows to capture only packets with Push and Acknolagment TCP flags (0000 0001 1000 = 24) – those were the only packets with payload of our interest*
and not port ssh	*Ignore any SSH traffic*

and not port 443	*Ignore any SSL traffic*
and not arp and	*Ignore any ARP messages*
and '(dst 172.31.129.9 and dst portrange 20001-20003)'	*We are only interested in monitoring connections which have been initiated by someone else and only with service ports. These rules help to filter out our own malicious traffic where we try to connect to other team's services*
or '(src 172.31.129.9 and src portrange 20001-20003)'	
-r [filename.pcap]	*Dump all collected packets into a .pcap file for further analysis*

In order to continuously run this command and collect the malicious traffic a bash script named *lunch-monitoring.bash* has been written to handle launching and stopping tcpdump every 3 minutes (dump traffic every tick) and handle .pcap file naming in order to avoid overwriting existing dumps and keep track of the process.

Once network traffic has been collected, analysis needed to be performed on all captured packets to identify any malicious payloads or stolen flags. Unfortunately, no python modules that would simplify .pcap file analysis were available on game machines, therefore we needed to find way to extract all necessary information with available tools. The easiest option was to utilize tcpdump once again since it provides functionality to read .pcap files and extract raw packets. *packet_parser.py* script was written to run tcpdump with -r parameter to read raw traffic information from .pcap file, decode it and parse to extract information from every collected packet. Our initial approach was to store all the information about packets in an SQLite database to make it easy to navigate and search for packets of interest. Packet information would be stored in the following fields:

timestamp	src. IP	src. port	dest. IP	dest. port	payload	FLG

where payload is the content of the message, and FLG is a binary indicator (0/1) which would be set to 1 if packet's payload contained string of form FLG[a-b0-9], indicating potential flag being sent out. The key idea was to monitor packets that contained flags and, in case one was detected, trace down the whole conversation between our service and adversary machine that resulted into dispatch of message containing a flag. This would allow us to see exactly which commands resulted into successful breaking into our service. The database idea, however, didn't work out since making python script talking to SQLite database turned out to be trickier than expected and wasn't finished before actual competition.

Final version of *packet_parser.py* simply processes .pcap files and stores all retrieved packets in readable version in txt file, which was then searched through for flags manually.

Original packet form (raw .pcap output)

```
 00:58:27.732933 IP 172.31.129.9.56341 > 172.31.129.18.microsan: tcp 14
E..B..@.@..S... ......N!...Z=.......Z......
.MtU....FLG0987654321
```

Parsed packet in human-friendly form

```
==================================== = = = = = = = = = =  =  =  ~         ~  -        -
|| Src. IP:         172.31.129.9
|| Src. port:  56341

|| Recv. IP:        172.31.129.18
|| Recv. port: 20003

|| Time:            00:58:27.732933

|| Data:            FLG0987654321
 ====================================== = = = = = = = = = =  =  =  ~      ~  -        -
```

- **Vulnerability detector**: a Python script is created to find out possible vulnerabilities in our services. In this script, lists of keywords that might become vulnerabilities (e.g. strcpy in C and $_GET in PHP) are defined and used to scan the source code. This detector helps us locate the unsafe features in a short time for further analysis.

- **PHP decoder**: In CTF 3, the PHP files are encoded with base64_decode and str_rot13 for several iterations. Therefore, we prepare a Python script to decode repeatedly until it's readable for human.

- **Shellcode**: the shellcode idea was thought out, but as Adam advised if you are already in a backdoor or have direct access, there is no need for the shellcode. With the shellcode, we based it from our last project where we used shellcode in several binary attacks. In this PCTF we only extended those ideas. However, once we realized we could gain direct /bin/sh access via an exploit, there was no need to pop a shell. We still created a couple of strings we could use in case that option of getting /bin/sh access wasn't available during the PCTF.

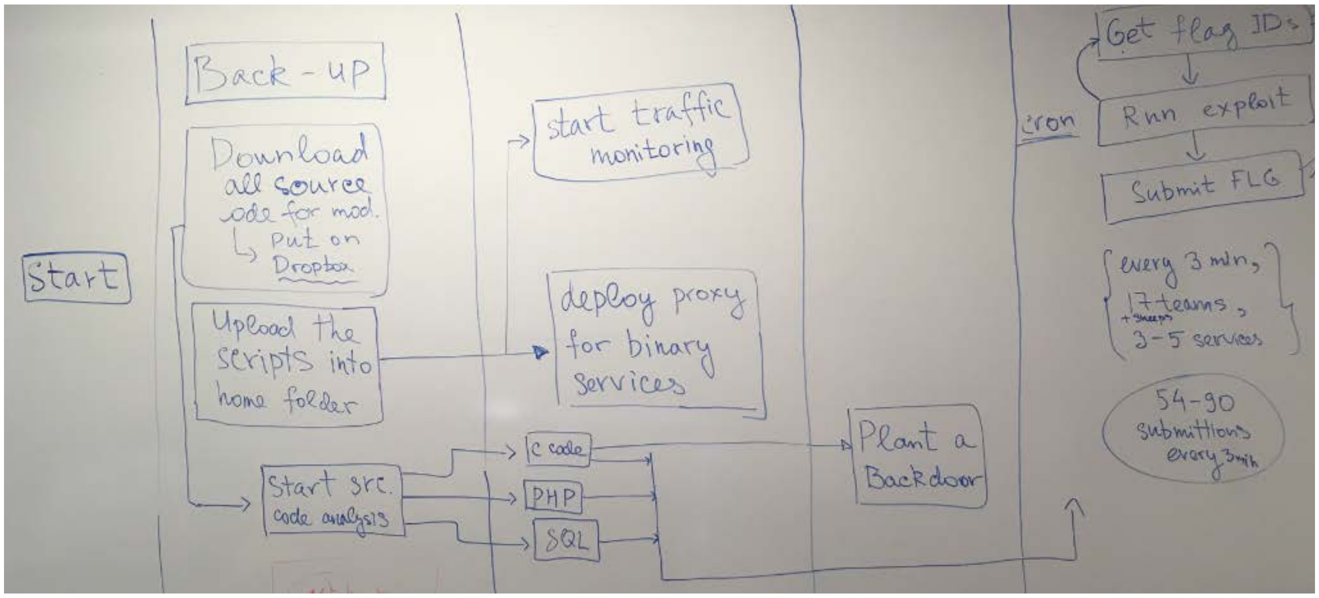

Start
Back-up
Download all source code for mod.
Put on Dropbox
Upload the scripts into home folder
Start src. code analysis
C code
PHP
SQL
start traffic monitoring
deploy proxy for binary services
Plant a Backdoor
cron
Get flag IDs
Run exploit
Submit FLG
every 3 min, 17 teams, 3-5 services
54-90 submittions every 3min

Figure 3. Game Day Plan

Why we chose it

Our thought process, we wanted to cover as many exploits we've touched on this semester, which will allow us to cover our bases defensively and give us the opportunity to attack if we could gain a backdoor or use the reflector an arsenal or journeyman tool to be successful. If the PCTF was going to be anything like CTF3, then our tools could serve a purpose. Figure 4. shows a capture from our initial brainstorming meeting.

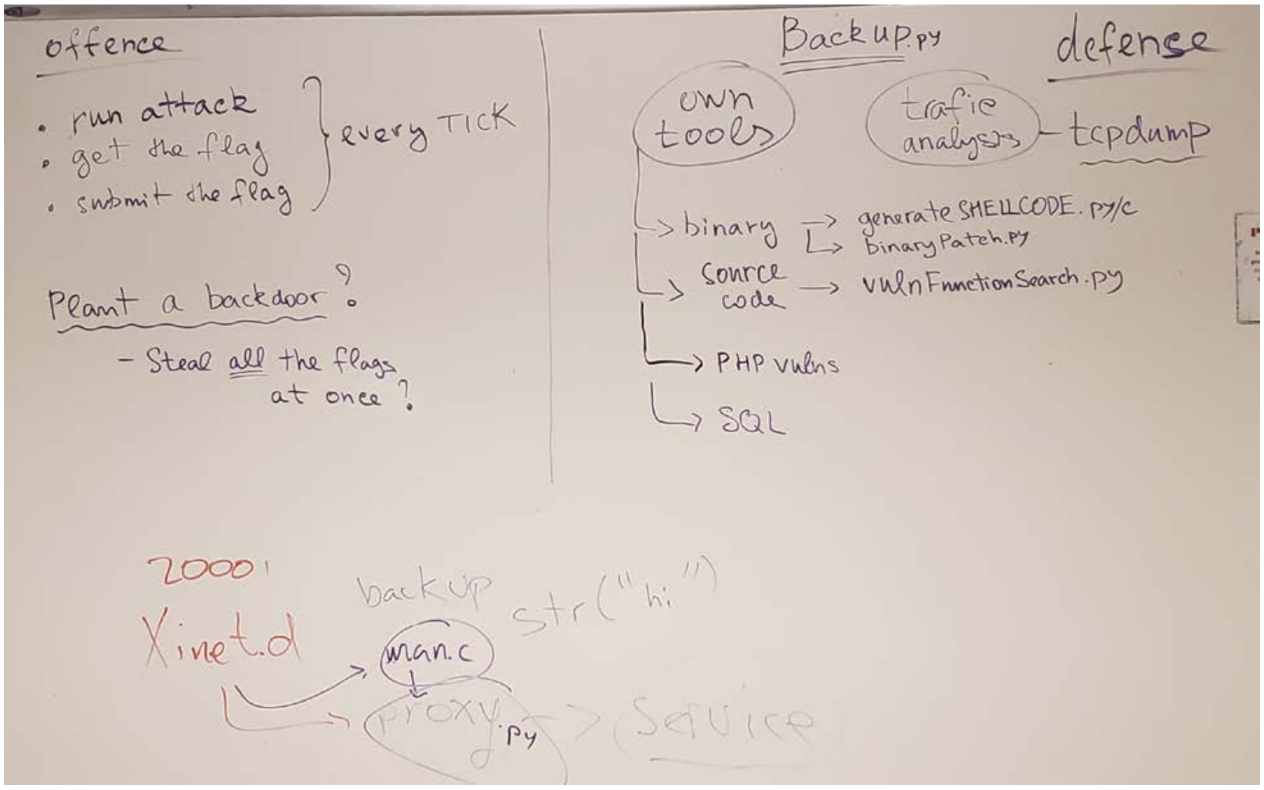
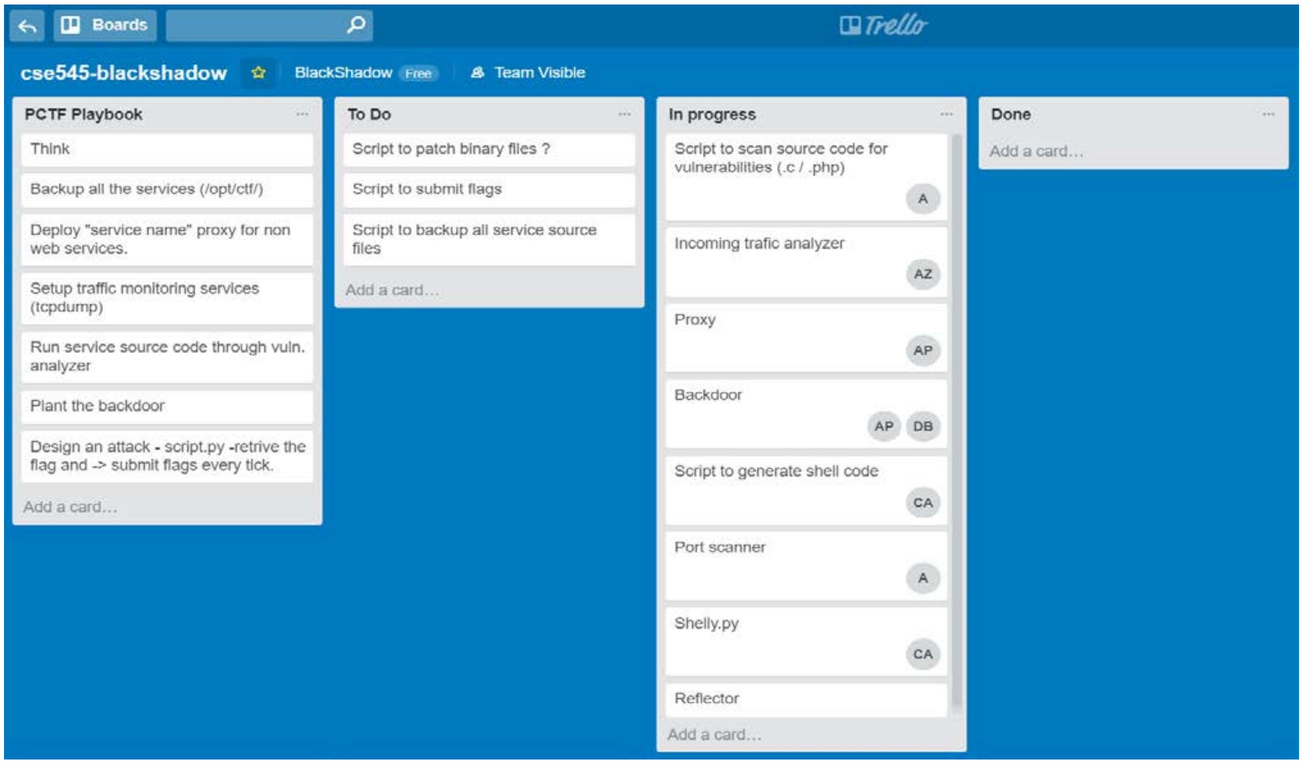


Figure 4. Brainstorming

How we implemented it

Since we had different areas to cover, each of the team member was given a tool to write up. We all provided backup to each other and worked every weekend to accomplish our tasks. We set up a Github repository for revision control and keeping everyone up to date. A Trello board (Figure 5) was also created to manage the preparations that should be done before the game day.

Albert wanted to work on proxy and the reflector - the proxy will sit between xinetd daemon and the service(s) it is communicating with so we could actively control incoming connections and the reflector could broadcast any attacks to other teams using our controls, so any successful attack on us could possibly also mean we could use the same process to exploit other teams. Debarati was tasked with adjusting the backdoor since she is our C expert. Gaining backdoor access could allow us to do as we intended based on the effective uid of the service we were running through. Avinash investigated patching current binaries and exploits and coming up with blacklisted items to look out for. Ting worked on the PHP and SQL exploits and defenses since he could make sense of the web insecurity section we were covering. Artem worked on analyzing and breaking down the tcpdump into easily readable format so we could make sense of how other teams were exploiting us and use it defend ourselves or use their steps to attack others. Cephas was tasked with writing shellcode and providing backup to others to help brainstorm their attacks or defenses.



PCTF Playbook
- Think
- Backup all the services (/opt/ctf/)
- Deploy "service name" proxy for non web services.
- Setup traffic monitoring services (tcpdump)
- Run service source code through vuln. analyzer
- Plant the backdoor
- Design an attack - script.py -retrive the flag and -> submit flags every tick.
- Add a card…

To Do
- Script to patch binary files ?
- Script to submit flags
- Script to backup all service source files
- Add a card…

In progress
- Script to scan source code for vulnerabilities (.c / .php) A
- Incoming trafic analyzer AZ
- Proxy AP
- Backdoor AP DB
- Script to generate shell code CA
- Port scanner A
- Shelly.py CA
- Reflector
- Add a card…

Done
- Add a card…

Figure 5. A view of the Trello board

How well the tools helped

Since the PCTF environment was a bit different than the CTF3, we stumbled right off the blocks. Seems our backdoor tool is the one that came in handy once we were able to exploit the backup-child service. Later in the session we were able to exploit securedb, but it was too late at that time to make a difference. We were scared to patch some of our services, since from experience in the last CTF it crippled us for a few hours when we patched backup, so although we were able to identify the exploits and wrote a script, we never applied it.

The proxy tool along with the services that were coupled with this tool ended up not working. Different configurations were attempted during the PCTF, but it was decided to remove the proxy in order to continue getting the points that were given for having the services up. This was because at this point we were falling behind and in around 16th place. The proxy was causing some sort of issue with the way it was replying and a guess would be it was due because of some sort of formatting with whitespace or newlines, an extra was either used or omitted. This wasn't thoroughly tested because there was no way to know what the bots were going to input or what they were expecting since CTF3 wasn't running ticks anymore by the time this tool was completed.

Using the traffic analysis tool, we were able to intercept someone's attempt to exploit us in the backup-child service. This was then taken and used on other teams to eventually lead to full shell access on their machines. From here auto exploit script was attempted to automate this exploit and submit flags to everyone but it was facing its own issues. At this point the pressure was on because what seemed like 5 minutes was actually about an hour now. So, having shell access to these machines it was known that the backdoor script to automatically get flags was working and wasn't going to require any modifications based on the service because it just connected to the backdoor. At this point half the members on the team were dedicating to install these backdoors on every machine they could. About 15 machines were exploited and the script to talk to the backdoor in order to find the flags was used, bringing us to 6th place.

The outcome faced wasn't what was planned but this shows us that when pressure is applied and we are trying to race the clock tools that require the least modifications will prevail. This showed that we did not perform enough testing on our tools before entering the PCTF.

What could be done to improve the tool

The keyword list for vulnerability detection can be extended a lot especially for the web security. Also our detector simply catches the keywords and displays the line numbers. We still have to analyze the code manually. Automation features such as using a script (Python or Bash) and utilizing regular expression can be introduced into our detector to increase the precision as well as to eliminate those safely-called functions.

With enough time, we could automate the steps to continuously exploit other teams once we were certain an exploit existed. Such as the backdoor exploit we successfully pulled off. Seemed to be more manual in the steps, once Albert found the exploit, Cephas and Debarati jumped in to help him exploit other teams, while other teammates focused on other exploits and defenses.

Testing could have prevented the proxy from making our services come down. This was not done because of the lack of time between the date of completion and the final PCTF. Little testing was done locally and even by monitoring traffic but there was something that caused the service to be marked as down essentially having us remove this proxy all together. It is thought that this occurred due to a formatting issue because the firewall, reflector, and verify flag were turned off making this proxy just forward traffic back and forth and the service was still marked as down coming the new tick.

Looking back at actual competition, we realize that traffic monitoring tool was indeed useful and allowed us to catch and reverse-engineer an exploit for “backup-child” before we were able to figure it out on our own. The database integration would be a cool add-on that could add more automation to our sniffer and give more flexibility.

Lessons Learned

Although pulling resources to the backdoor attack didn’t make much of a difference in the end, it could have cost us some points because we pulled aside resources to assist in attacking others as opposed to having an auto process. It was still fun finding out some teams were still vulnerable to the attack.

Another thing we found when exploiting securedb is: even the PHP file already initialized variable $isAdmin as false, we still could overwrite it by providing the value in a GET request. The reason is $isAdmin is declared in a global scope so that it fails to prevent the parameter creation attack. We should never leave this kind of “testing” feature in our deployed program.

Conclusion

We felt we did well. Although we didn’t place in the top 3, we still felt were able to exploit other teams successfully. Our proxy failed to work, it started off and died in the first few minutes. Securedb, we almost got a hang off, but it was at the end of the pctf and would require more ticks than we had left to successfully have it up and going. The file analyzer came in handy and helped us see how we were being attacked and how we could patch it. So in all for only having the backdoor, we were able to come in top 5, but eventually got bumped in the last few ticks to 6th place. Giving a big shout out to the team, and special thanks to Albert, and Ting who carried us through most of our CTFs including this PCTF.