1 SEC Consult – Cyber Security Challenge Austria / CTF Tips & Tricks

This article is intended to give useful tips and tricks for participants of the Cyber Security Challenge Austria, however in general, it can be seen as a guideline to Capture-the-Flag (CTF) competitions. The first chapter is for beginners who don't know where they can acquire the required skills to participate and how to get started. The later chapters give some hints for various categories in CTFs. Please note that the tips range from basic tips to more complex ones like "how to solve hard binaries" or "how to win attack-defense CTFs" (which is very likely not required for Cyber Security Challenge Austria). Please also note that the article has a strong focus on the binary / exploit category because that's the authors main category.

1.1 How to get started?

This chapter is for people who want to do more in the field of IT security, but who don't know how to get started and where the required knowledge can be learned.

The best source of information are books that summarize the most common attack techniques and protections against them. Since the entire topic of IT security is very extensive there is not a "one-size-fits-all" book which lists all attacks in-depth. Instead, there are standard books which cover specific topics. The following list is not exhaustive, but tries to list the best books (in the authors opinion) for the respective topics:

- Web
  - The Web Application Hackers Handbook by Dafydd Stuttard
    - The standard book on web attacks which covers lots of different topics.
  - SQL Injection Attacks and Defense by Justin Clarke-Salt
    - A very good and detailed explanation of SQL injections.
- Reverse Engineering
  - Practical Reverse Engineering by Bruce Dang, Alexandre Gazet, Elias Bachaalany
    - A good introduction to x86/x64 assembler and reverse engineering.
  - The IDA Pro Book by Chris Eagle
    - The best introduction to IDA Pro (the disassembler which you want to use) together with a great introduction to reverse engineering.
- Exploit Development (Memory Corruption)
  - Shellcoders Handbook by Chris Anley
    - The standard book on the topic of memory corruptions and exploitation.
  - If German books are preferred, "Hacking: Die Kunst des Exploits" by Jon Erickson is a possibility. Another option is "Buffer Overflows und Format-String-Schwachstellen" by Tobias Klein.

    - There are also lots of good videos on the topic of exploit developing (which maybe give a better introduction to the topic).

- Cryptography
    - Serious Cryptography by Jean-Philippe Aumasson
        - By far the best book on the topic.

There are also lots of very good free video series available on the internet which serve as a great introduction:

- LiveOverflow: https://www.youtube.com/channel/UClcE-kVhqyiHCcjYwcpfj9w/playlists
    - A good introduction especially to binary hacking and lots of solutions for CTF challenges in compact videos.
- GynvaelColdwin: https://www.youtube.com/user/GynvaelColdwind/videos
    - Live streams of a very experienced CTF player. You can learn a lot by watching him solve challenges.
- IppSec: https://www.youtube.com/channel/UCa6eh7gCkpPo5XXUDfygQQA/videos
    - Live walkthrough through several vulnerable systems and how you can hack them.
- Murmus CTF: https://www.youtube.com/channel/UCUB9vOGEUpw7IKJRoR4PK-A/videos
    - Live streams from CTF and real-world bug hunting.
- Securitytube Megaprimers: http://www.securitytube.net/groups?operation=view&groupId=4
    - Securitytube is a website like YouTube but just for security related videos. The owner of the website created lots of very good videos which he published as free "megaprimers". The assembler, buffer overflow and WLAN security megaprimers are a very good (if not the best) introduction to the topic, however, most of them are no longer available. Newer videos became commercial, but maybe the old (free) videos can still be found somewhere on the internet.
- OpenSecurityTraining: http://opensecuritytraining.info/Training.html
    - Trainings which are freely available (slides, videos and source code). The x86 videos together with exploits 1 and exploits 2 give a good introduction to exploit development. "Life of binaries" gives a good introduction to the PE / ELF file format (e.g. to development or to understand packers / crypters).
- Reversing for Newbies from Lena: https://tuts4you.com/e107_plugins/download/download.php?list.17
    - These videos don't have sound and are already more than 10 years old but can still give a good introduction on reverse engineering.

If you read all the mentioned books and viewed all the videos and want to learn even more, a good starting point is phrack (http://www.phrack.org/issues/1/1.html) or uniformed (http://www.uninformed.org/ for older techniques).

If you are already familiar with the basics of all the above-mentioned categories and want to learn about the newest attack techniques, you should start to use Twitter. Following the correct people on Twitter will

help you to keep up-to-date on the latest attack techniques and protections. This will most likely not help you in competitions like the Cyber Security Challenge but will help you in real-life scenarios.

Another good source of information is reddit in the subreddits netsec and ReverseEngineering.

Still not enough? Then go to the bug tracker of google project zero (https://bugs.chromium.org/p/project-zero/issues/list?can=1&redir=1) or HackerOne (https://hackerone.com/hacktivity?sort_type=upvotes&filter=type%3Aall&page=1&range=forever) and read the writeups.

Reading writeups from past CTF events can also help you a lot. You can find writeups on CTF-Time (https://ctftime.org/writeups) by selecting an event and then checking if writeups are available. If you are not already in a CTF team you can use this site to find a CTF team close to you. Examples of good CTFs with interesting challenges are iCTF, SECCON, PlaidCTF, RuCTF, Defcon CTF, Chaos Computer Club Congress CTF, …

If you want to practice the learned techniques and experiment with them, you can download / setup a virtual machine locally. There are lots of (intentionally) vulnerable web applications available for learning purposes. For example:

- OWASP DVWA (Damn Vulnerable Web Application)
- OWASP WebGoat
- OWASP Juice Shop
- OWASP Multillidae 2
- OWASP NodeGoat
- OWASP Railsgoat
- OWASP DVWS (Damn Vulnerable Web Sockets)
- Metasploitable
- XVWA (Xtreme Vulnerable Web Application)
- DVRW (Damn Vulnerable Router Firmware)

The easiest way to get started is to download the "web security Dojo" as a .ova file. This file can be imported as a virtual machine (e.g.: in virtualbox) and already contains most of the above-mentioned vulnerable applications. If you are stuck in one of the challenges you can always try to search the application name on YouTube, you will find lots of walkthroughs.

Here are some other websites where you can practice your newly acquired skills online:

- https://www.hacking-lab.com/
- https://exploit-exercises.com/
- http://pwnable.kr/
- http://io.netgarage.org/
- http://overthewire.org/wargames/
- https://www.root-me.org

- https://www.vulnhub.com/
- https://www.hackthebox.eu/
- http://xss-game.appspot.com/

Another useful link:

- https://github.com/apsdehal/awesome-ctf

1.2 General Tips

- In general, you should not use tools to solve the online challenges. Lots of challenges contain protections to protect against such tools. For example, if you identify a SQL injection, try to solve it manually and don't just run SQLMap against it. You will also learn a lot more!
- If you are stuck in a challenge and don't know how to solve it, try to read writeups from other challenges / events in the same category. They may help you to find a different approach which you didn't think of before.
- Don't go into too much depth too soon! Don't focus on a single vulnerability which you identified. It's common especially for attack-defense CTFs to put multiple vulnerabilities into one application where only one of the vulnerabilities is (easily) exploitable. Trying to exploit one of the other vulnerabilities is a common mistake in CTFs and can waste a lot of time. Example: The pokemon challenge from iCTF 2017 contained at least five different vulnerabilities (integer wrap, two use-after-free bugs, one bug which allowed to move memory around and an injection via a space character). Only one was easily exploitable (the injection).
- Have local test systems already set up. For example: When identifying a hard SQL injection, it often helps to re-create the SQL query in a local running SQL server and writing the query / injection there. Having qemu running for different architectures will also safe you a lot of time.
- If you want to win a competition automation is key. To win you must solve the challenges faster than other competitors and you can achieve this by automating the most important tasks. Example: Some years ago, Defcon CTF qualification had a challenge with a backdoored openssh server. Participants got the openssh server together with a file containing "garbage". The task was to reverse engineer the binary, identify the backdoor and how it stores the passwords obfuscated along with other random data in the "garbage" file. Our automation scripts ran XOR with all possible values and combinations against the file and after that called strings on it. If the number of found strings was higher than the average, this was reported. Using this technique, we solved the challenge as the first team in under 5 minutes. Other example: Often the task from a reverse engineering challenge is to find the correct password / serial which then prints a success message. The correct input can be identified by reverse engineering the involved algorithms which can be very time-consuming. A faster solution is to count the number of executed instructions (e.g. with valgrind, PIN or DynamoRio) which is often possible. If the number of executed instructions differs if the first character is correct, you can bruteforce the password byte-by-byte by checking for which input the number of executed instructions is higher. We already solved several challenges using this technique in under one minute. Other techniques can be to run a symbolic execution engine or start to fuzz the binary. Please note: Running offline tools is no problem because they don't DOS (denial-of-service) the challenge.
- Learn a scripting language like python. Writing a quick attack automation script is a mandatory

skill in attack-defense CTFs.

- In an attack-defense CTF where the teams attack each other, defense is often more important than offense. Having the service up and running should have highest priority, after that patching / protecting the service, and only then the development of an exploit. It's often faster to "steal" the exploit from another team from the network dumps or logs than to develop an exploit from scratch. Lots of services can additionally be hardened before the vulnerability itself is even identified. (Note: do this with care because in some CTFs this is considered cheating). For example: Protections like ASLR and DEP can be enabled on the system / binary if not already done. The binary can be started with a different heap implementation (LD_PRELOAD) which protects against heap flaws. Instrumentation frameworks like DynamoRio can be used to protect against other vulnerability classes like out-of-bound writes or uninitialized variables (DrMemory). Please bear in mind that network related tricks like checking the TTL (Time-To-Live) field to detect if traffic originates from the game server or other teams (to block other teams) is forbidden in all CTFs and you won't learn anything by using these techniques. The major goal of the competition should be to learn new stuff and not to win.

1.3 Category: Web-Vulnerabilities

When auditing a web application, you should use a proxy between your browser and the website. The most common tools for this are Burp and OWASP ZAP. Some people use browser extensions to modify requests, however, you will see that using a proxy gives you way more flexibility and you will be much faster. On YouTube you can find good tutorials on these tools by searching for one of the previously mentioned vulnerable applications together with the tool name (e.g.: "Multillidae burp"). You should get familiar with the repeater, intruder and decoder tab of Burp.

- Here is a (not comprehensive) list of things to check in web applications (if you don't know some of the vulnerability classes use google to find explanations):
  - Read the source code / comments
  - Check for common hidden files / folders (.git, .ssh, robots.txt, backup, .DS_Store, .svn, changelog.txt, server-status, admin, administrator, …)
  - Check for common extensions (Example: If you see a index.php file, check index.php.tmp, index.php.bak, and so on)
  - Play with the URL / parameters / cookies (Example: If you have a page with index.php?role=user try to change it to index.php?role=admin).
  - Get familiar with the website, it's functionalities and features before starting an in-depth analysis.
  - Try to map the full attack-surface of the website! Some vulnerabilities are hidden deep in hard-to-reach functionalities.
  - Test for the most common vulnerabilities like SQLi (SQL Injection), XXE (XML Entity Injection), Path Traversal, File Uploads, Command Injection, Cookie Tampering, XSS (Cross-Site-Scripting), XPATH Injection, Unserialization bugs, Outdated software, CSRF (Cross-Site-Request-Forgery), SSRF (Server-Side-Request-Forgery), SSTI (Server-Side Template Injection), LFI/RFI (Local-File-Inclusion / Remote-File-Inclusion), Flaws in Session Management or Authorization Flaws, the randomness of the cookies, and so on. If

you don't know one of these vulnerability classes, just google them, you will find lots of good tutorials.

- If you come across a technology which you don't know, try to google security writeups for these technologies.
- Try special characters (`, ", {, ;, |, &&, \, /, !(), %…) in all input fields (GET- and POST-parameters and Cookies) and check for uncommon responses or error messages.
- To detect blind vulnerabilities (SQL injection, command injection, XSS, …) you can use time delays or requests to one of your web servers (check the access logs).
- If you identify a possible SQL injection, write your assumed SQL query inside a text editor together with the inputs.
    - Example: You assume that you have a SQL query like SELECT * FROM users WHERE name=INJECTION1 AND password=INJECTION2
    - Then you provide specific values for input1 and input2 and check if the query behaves as you except. For example, you can execute the query in a local data base and check if the response from the website is the same. For example, you can try to add a comment in injection1 and check if injection2 is really commented out. Or you add a multi-line comment start (/*) in injection 1 and a multi-line comment end (*/) in injection 2 and check if this really works. This way you can easily identify the order in the query (if input1 is checked first or input2). Moreover, you can try to inject OR and AND true/false statements to identify the binding in the query. If the injection is hard to exploit, you should first try to understand as much as possible about the query before exploiting it.
    - One of the first steps should be to identify which DBMS (database management system) is running. You can identify this via error messages or by doing special requests which are handled differently in every DBMS. For example, you can try different ways of string concatenation or different ways of comments. A good list of tests can be found in the book "SQL Injections Attacks and Defense" or in SQL injection cheat sheets which you can find via google. With MySQL you can also use an injection which creates queries like: SELECT * FROM users where username=admin /*!51011 1/0*/ --%20. The code in the comment will only be executed if the DBMS version is higher than the tested version number (51011 in the above case).
    - Don't forget that MySQL comments require a space after the two dashes. In GET parameters you can therefore inject "--+" (+ will be interpreted as space). Otherwise you can use %20 or # (don't forget to URL-encode it!).
    - In most cases you don't need spaces to exploit a SQL query. The exact techniques depend on the target DBMS, however, you can try multiline comments (in MySQL) like SELECT/**/1/**/FROM/**/USERS. Another technique is to use (and) to inject something like: "OR(1=1)OR'a'='. And another trick is to use || (for OR) or && for (AND). Example: '||'1'='1.
    - If the website shows SQL error messages and you have a binary SQL injection (which just reflects the outcome of the query with two states like "logged in" or "not logged in"), then you can dump the data via the error message (for example

with the extractvalue() function in MySQL). You can also try to use a second channel (google sql injection <dbms name> second channel).

- If you can provide a path or a filename to the website, you should test for path traversal vulnerabilities. If the application replaces the "../" with an empty string, you can try to bypass it by injecting the sequence two times, like: "…/./". If the "../" in the center gets replaced, the application will again work with "../". You can also try different encodings or other removed characters. Moreover, you can try to create or upload (e.g. via archives) a symbolic link.
- If you found a LFI (local-file-inclusion) vulnerability in a PHP website and you want to read the PHP scripts, you can use php-filter (you can't normally read .php files because the inclusion would try to execute the code instead of displaying it; with php-filter you can first base64-encode the content to display it): index.php?filename=php://filter/convert.base64-encode/resource=index.php

- Useful tools / Links:
  - JS Beautifier: http://jsbeautifier.org/
  - OWASP Cheat Sheets: https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series
  - XSS Challenge Wiki: https://github.com/cure53/XSSChallengeWiki/wiki
  - JavaScript data-flow analysis: http://www.fromjs.com/
  - SQL Injection cheat sheet: https://websec.ca/kb/sql_injection
  - Attacking Ruby on Rails: http://www.phrack.org/issues/69/12.html
  - HTML5 Security: http://html5sec.org
  - Burp HUNT plugin: https://github.com/bugcrowd/HUNT

1.4 Category: Cryptography

- The padding oracle vulnerability is very common in CTFs, you should know this vulnerability and how to exploit it. There are scripts which can automatize exploitation for you.
- If a MAC (Message-Authentication-Code) is implemented in a weak way with hash algorithms like SHA1 or MD5, then it can be vulnerable to so-called length-extension attacks. The problem arises if data is hashed with a secret key which is the first input to the hashing function. For example: If the application calculates MAC=md5(<secret><message>) the idea is that the attacker can't change the message, because then the MAC will be different. The attacker can also not calculate the updated md5-hash because the secret is not known to him. However, in this scenario the attacker can perform a so-called length-extension attack. That means if the attacker knows a valid MAC / message pair (but not the secret), the attacker can add more data to the end of the message and calculate the new MAC without knowing the secret value. For example: If the application protects SQL statements with this MAC algorithm and the valid input is user=user1234, you can change the input to user=user1234' or '1'='1 and re-calculate a valid MAC so that the query will be accepted. You can use the HashPump tool for this.
- Lots of crypto challenges are based on problems with RSA keys which use weak values. It's best to read old CTF writeups on crypto RSA challenges to learn more about this (use Google to find

some).

- Crypto vulnerabilities can often arise from weak random numbers. You should carefully check the way random numbers are created and if these are cryptographically secure.
- You should check if you can extract some information to break the crypto using a side-channel.
- A website / application should not store user credentials in cleartext in a database. Instead, hashing (with salt and pepper) should be used. If a user later wants to login on the website / application, he must send his password to the site with the login request. Then the website can hash the password and compare if the hash in the database matches. A programmer may think that hashing the password on the user-client is more secure because then the cleartext password is never sent over the network, however, this introduces a pass-the-hash vulnerability. The problem is that an attacker just has to steal the password hashes and then he doesn't have to break the hashes because he can authenticate to the website / application by just presenting the stolen the hash. The same vulnerability gets exploited in Windows Domains for two decades because the NTLM authentication protocol is vulnerable to pass-the-hash attacks. The correct solution would be to hash on the server and the communication should be protected by HTTPS / TLS.

Here is another useful website:

- https://www.cryptool.org/

1.5 Category: Reverse Engineering / Exploit Development

The most important tools for reverse engineering are a good disassembler like IDA Pro (the standard tool used by most people), Hopper, Binary or radare2 and a good debugger (on Linux GDB or remote IDA Pro debugging and Immunity Debugger or WinDbg on Windows).

You should use the disassembler to understand what the code is doing and then the debugger to verify your assumptions. The two most important skills in reverse engineering are:

1. Finding the correct code location with the "important code". In small CTF tasks this is not as important because most of the time the binary is so small, that it only contains "the important code". However, in real-world applications you typically deal with millions of assembler code lines and then it's crucial to quickly find the important code. You can typically do this by following the input (memory breakpoints on input or a taint engine), setting breakpoints on specific functions (e.g. the read-data-from-network function and then checking the callstack), or with cross-references to strings (IDA Pro View Strings, then double click the string and use the x-hotkey to find references). Another option is differential analysis. In differential analysis you execute the application two times, one time with the target behavior which you want to analyze and one time without. In both executions, you measure the executed code and then subtract the generated code coverage files from each other. The result will be the code which you didn't trigger the second time and therefore just the behavior which you want to analyze. To extract this code coverage information you can use PIN or DynamioRio (e.g.: just run the command: drrun.exe -t drcov -- ./target_application arg1 arg2). After you created two coverage files you can subtract them inside IDA Pro with the Lighthouse plugin (https://github.com/gaasedelen/lighthouse).
2. The second important skill is to learn to comment and name the code which you analyzed. When teaching reverse engineering I very often see that people don't "put the gained knowledge" back

into the IDA Pro database. The result is that later code will be much harder to understand. Example: The following image shows code from the "chat" binary from SECCON 2016 ctf (the hardest in the exploit category):

```
1  void __fastcall remove_user(__int64 a1)
2  {
3    _QWORD *v1; // ST28_8
4    _QWORD *v2; // ST28_8
5    void *v3; // ST28_8
6    __int64 i; // [rsp+18h] [rbp-18h]
7    _QWORD *ptr; // [rsp+20h] [rbp-10h]
8
9    for ( ptr = *(_QWORD **)(a1 + 8); ptr; ptr = v1 )
10   {
11     v1 = (_QWORD *)ptr[18];
12     free(ptr);
13   }
14   for ( i = tl; i; i = *(_QWORD *)(i + 144) )
15   {
16     if ( *(_QWORD *)(i + 144) && *(_QWORD *)(*(_QWORD *)(i + 144) + 8LL) == a1 )
17     {
18       v2 = *(_QWORD **)(i + 144);
19       *(_QWORD *)(i + 144) = v2[18];
20       free(v2);
21     }
22   }
23   if ( tl && *(_QWORD *)(tl + 8) == a1 )
24   {
25     v3 = (void *)tl;
26     tl = *(_QWORD *)(tl + 144);
27     free(v3);
28   }
29   free(*(void **)a1);
30   free((void *)a1);
```

As you can see, it's very hard to understand what this code does. Therefore, you should go to simpler code and try to improve your database first (e.g. with simple functions like print-stats, save-settings, load-settings, …). Here is the same code from my IDA Pro database from during the CTF:

```
void __fastcall remove_user(user_data *user_address)
{
  tweet_data *next_tweet; // ST28_8
  tweet_data *tmp; // ST28_8
  tweet_data *tmp2; // ST28_8
  tweet_data *iterator_public_tweets; // [rsp+18h] [rbp-18h]
  tweet_data *user_tweet_iterator; // [rsp+20h] [rbp-10h]

  for ( user_tweet_iterator = user_address->tweet_data_list_head; user_tweet_iterator; user_tweet_iterator = next_tweet )
  {
    next_tweet = user_tweet_iterator->next_tweet_entry;
    free(user_tweet_iterator);                  // Only DirectMessages which the user received (not send!)
  }
  for ( iterator_public_tweets = g_public_tweets_head;
        iterator_public_tweets;
        iterator_public_tweets = iterator_public_tweets->next_tweet_entry )
  {
    if ( iterator_public_tweets->next_tweet_entry
      && iterator_public_tweets->next_tweet_entry->sender_user_address == user_address )
    {
      tmp = iterator_public_tweets->next_tweet_entry;
      iterator_public_tweets->next_tweet_entry = tmp->next_tweet_entry;
      free(tmp);                                // Public messages
    }
  }
  if ( g_public_tweets_head && g_public_tweets_head->sender_user_address == user_address )
  {
    tmp2 = g_public_tweets_head;
    g_public_tweets_head = g_public_tweets_head->next_tweet_entry;
    free(tmp2);
  }
  free(user_address->username);
  free(user_address);
```

The difference is that I re-named variables in IDA Pro to their purpose with the hotkey n, created structures (in the struct tab) and assigned the type to variables via right click. Moreover, I changed 1-byte values to arrays (undefine the variable first with the hotkey u, then select all the bytes and click edit->array). Doing these three steps again and again (and adding comments with the semicolon hotkey in the assembly or with the slash in the decompiled C-code) will help you a lot in reverse engineering!

In case you are unsure which disassembler / debugger you should start with I put together some notes:

- Disassembler: The best choice is IDA Pro, however, the commercial license is not cheap. Since version 7 you can also use the free non-commercial license for CTFs, so there is no reason to not use IDA Pro (except if the CTF binary is for an architecture not included in the free license). Hopper and Binary Ninja are cheaper alternatives. Radare2 is a community project with lots of cool features, however, to use it you must know lots of shortcuts for the different commands which may be hard to start with. Therefore, I would recommend IDA Pro.
- Debugger Linux: The simplest solution is to use IDA Pro remote debugging (in the IDA Pro install folder you can find remote servers which you can start on Linux and then connect with IDA Pro to the opened port). This is especially useful when you try to understand the binary and you are searching for vulnerabilities in it (because you have a GUI). The other option is to use the command line tool GDB (maybe with plugins like GEF). After you identified the bug and start to develop an exploit you should switch to GDB because GDB is much better suited for this task. You can very quickly restart the application inside GDB and automate lots of steps which helps a lot in exploit development.
  - When working with GDB I do not enter the commands in the GDB prompt, instead, I always write them into a commands file which I pass to GDB with the "-x" argument. This has a big advantage: I can very quickly restart the application and execute exactly the same

breakpoints and commands again and again.

- Moreover, I use the "commands" command in the command files. For example, my command file has entries like:

```
set pagination off
bp *0x1234567
commands
silent
print "Start of function XYZ"
p /x $rdi
p /x $rcx
x /20xg $rsp
c
end
```

  - Explanation: First we set a breakpoint, after that we use "commands" to tell GDB to always execute the following commands at the breakpoint. Silent tells GDB to not display a message when the breakpoint hits, the print is a message for me to know which function was triggered, then the commands which I want to execute (e.g. display arguments) and at the end a "c" for continue to continue from the breakpoint and the end ends the commands-command. I create these command files with an IDA Pro plugin. Inside IDA Pro I set breakpoints on interesting functions, then execute the plugin which creates such command files. Then I execute the commands file on Linux with GDB and see exactly when the important functions are called with which arguments. This workflow is also extremely useful when exploiting heap overflows because you can visualize the current heap layout at every step.
  - An even better option is to write small GDB plugins in python because then you can automate GDB with a scripting language (and the log messages look much better). You can use GEF or PwnDbg as templates, it's really simple to write such a plugin!

- Debugger Windows: On 32-bit I prefer Immunity Debugger, on 64-bit or when reversing Windows functionality you want to use WinDBG. If the binary contains anti-debugging functionality you can try the !hidedebug plugin from Immunity Debugger or switch to Olly Version 2 (which has better debugging hiding functionality). If all that fails, you can try Windows remote debugging with WinDbg because this typically hides very good from anti-debugging techniques. However, in most cases you can also just use the debugger of IDA Pro (which is again the simplest solution). When developing exploits, you may want to switch to Immunity Debugger because the mona.py plugin which is available there helps a lot. WinDbg is mainly used to deal with Windows functionality to resolve .pdb files from Microsoft and to work with Windows libraries.

Here are some more hints for CTFs:

- Check for common vulnerabilities like: Buffer Overflow (Stack, Heap, BSS, …), Use-After-Free, Double-Free, Type-Confusion, Command Injection, Integer Flaws (Wrap Arounds and Sign Flaws), Format-String Vulnerabilities, Path Traversal, SQL injection, XXE (XML Entity Injection), …
- You should first try to get an overview about the application and it's features. That means you should not start to understand every aspect and function of the binary in-depth. Instead, go quickly over everything to get an overview. After that, go into depth at the most important locations.

- It's often faster to check the dynamic behavior of the application instead of trying to understand it statically. If I come across a "hard-to-understand" function / code / instruction in IDA Pro, I switch to the debugger and execute the function with different inputs and check how it behaves.
- One of the first things which you want to do is to run "file" on the binary. It gives you important information like the architecture, if it's dynamically or statically linked and if symbols are stripped or not. Dynamically linked binaries and unstripped binaries are simpler to analyze (in a statically linked binary which is stripped you don't know which code is application code and which is code belonging to libraries. In such a case you can use the FLIRT signatures from IDA Pro (or the Diaphora plugin). If a binary is statically linked it can also be an indicator that it may require ROP (return-oriented-programming) to solve (statically linked binaries are much bigger and ROP requires lots of gadgets in the binary if you can't leak the address of a library like libc).
- One of the next things which you want to do is to check which protections are enabled for the binary. You can use the checksec-tool for this. This feature is also integrated in many GDB plugins like GEF (you start gdb, load GEF via source gef.py and then execute the checksec command).
- If it's an attack-defense CTF, you should check if ASLR is enabled on the system: cat /proc/sys/kernel/randomized_va_space; If it contains a 0, it's disabled, then should write a 2 into the file to enable it.
- GEF is a very useful plugin for GDB for exploit development with lots of useful features (https://github.com/hugsy/gef). You should definitely check all the supported features.
- In some CTFs you must do ROP inside libc to exploit the binary. If it's an attack-defense CTF you can find the libc on your own machine, however, in non-attack-defense CTFs the libc binary must often be leaked via other challenges. For example, if you managed to get RCE (remote-code-execution) in the web-category, you can use this RCE to download libc from the system and then use it in the binary/exploit category to develop an exploit. There are also online-databases for many versions of libc (https://github.com/niklasb/libc-database and https://libc.blukat.me/). Moreover, some challenges require that an information leakage is used to download the libc via the vulnerability. In such a case you don't have to implement the logic yourself, you can use the DynELF feature from the PwnTools framework instead.
- You should start to use PwnTools and develop your exploits with this framework (https://docs.pwntools.com/en/stable/).
- In attack-defense CTFs it's often simpler to steal attack-payloads from enemy teams than developing an exploit yourself. For such situations it's useful to have a script ready which can take a PCAP stream number, extract the sent data and automatically create a PwnTools script which replays the data. Then you can easily steal the exploits from other teams by calling this script on network dumps and just modifying the parts in the exploit which depend on the returned data (e.g. a pointer leak to bypass ASLR).
- If ROP is required to exploit the binary (because the protection DEP (Data-Execution-Protection) / NX (No-Execute-Bit) is enabled) and you know the libc address, you don't have to write the full shellcode in ROP. The typical task of shellcode is to call the execve-syscall with /bin/sh as argument and for this you must write /bin/sh in ROP to some memory location. In many cases you can skip this step because /bin/sh can already be found inside libc and therefore you can pass the address of /bin/sh from the libc directly to the execve-syscall. Moreover, there is also something which is called a "one-gadget". That is a single address inside libc and if you overwrite the return-address or function pointer with this address it will spawn /bin/sh and therefore you don't have to waste

your time developing a ROP chain (https://github.com/david942j/one_gadget).

- If you need to exploit a memory-corruption vulnerability (buffer overflow, use-after-free, …) you should first check which characters are forbidden in the input. For example, the null-byte or spaces are often forbidden. You can easily test this by providing all values from 0 to 0xff as input and check if they are also stored in memory. If you then for example find the following bytes in memory: 00 01 02 03 04 05 06 07 08 09 [garbage]. Then you know that either 0x09 ends the input (and is therefore not allowed) or 0x0a is not allowed. Then you give again the sequence as input but omit the bad characters until everything is in-memory. Doing this before exploiting the challenge can eliminate sources of errors early. You don't want to write ROP chains and then see that they don't work and search for the reason (because one of the bytes from the address was not allowed).
- Many challenges are nowadays especially written to be solved with angr. Angr is a binary analysis framework for concolic execution. You should know how and when to use it. (https://docs.angr.io/)
- PONCE (based on the Triton framework) is a useful IDA Pro plugin which can do nearly the same as angr (https://github.com/illera88/Ponce). If you know which code you must reach in the binary (e.g.: you need the correct serial to reach the "success" message; therefore, you want to reach the code block which prints "success"), then you can use PONCE to calculate inputs to reach this code. This works by marking the input as symbolic inside the debugger of IDA Pro and when you would take the wrong path you can use PONCE to calculate an input to take the other path (which goes to the "success" message"). Angr is generally more flexible, however, PONCE is simpler to use because it's integrated into the GUI of IDA Pro.
- If you search for strings in the binary with IDA Pro, make sure to also search for Unicode strings (right click in the strings view, setup, check Unicode strings). You can also pass different encodings to the Linux strings-command via the "-e" flag!
- If you have to exploit a heap overflow, you should check out the How2Heap repository by shellphish: https://github.com/shellphish/how2heap It lists different heap attack techniques together with links to previous CTF writeups where the techniques were required. You can also learn a lot by reading these writeups. The GDB plugins GEF and libheap can also help a lot when developing heap exploits.
- If your target binary reads from stdin and you pipe your exploit into it, don't forget to keep stdin open! Example: You try to spawn a shell by injecting shellcode / ROP which calls system("bin/sh"). If your exploit works, the application will spawn a shell. However, if you start the application like: "python exploit.py | ./vulnerable_application" the shell will be opened, but immediately close again and therefore it will look like your exploit failed! The correct command should be "(python exploit.py;cat) | ./vulnerable_application" because cat keeps stdin open. This is a very common pitfall which I saw with many different people in many challenges (also myself)! Don't forget it or you will lose time searching for a problem which doesn't exist!
- If you develop your exploit inside the GDB debugger and manage to spawn a shell, make sure to restart GDB afterwards. A common pitfall which I saw with many students is that they spawned the shell but didn't see the GDB message that /bin/sh was executed. If the students then wanted to test the exploit again, they just re-run the application in GDB (with "r" for "run"), however, since GDB now executed /bin/sh this command will start /bin/sh and not the target application. In such a case you must quit and start GDB again.
- Try to always go as far as you can (but keep on be legal side) – you will learn a lot more! Example: Last year CSC Austria contained a challenge where the code flow could be modified (by overwriting

a function pointer). Moreover, the binary contained a function named "print_flag". The solution was obvious: overwrite the pointer to point instead to the print_flag function and read the flag from the returned data. The address of the print_flag function was always the same because the binary was not compiled as PIE (Position-independent-executable) and therefore addresses were not randomized by ASLR (Address Space Layout Randomization). So, after getting the flag I could have stopped, however, such a vulnerability can nearly always be turned into full code execution (the binary was running on a remote port), so it's a nice challenge to also exploit this one. I don't have my writeup anymore, so I will explain my steps from my memory. There were some problems when trying to get RCE: First, the binary was compiled with NX support, therefore shellcode can't be injected and therefore ROP must be done. For ROP, at least some gadgets are required, however, the binary was dynamically linked and therefore very small which means that there were literally no gadgets. Therefore, the ROP chain must be built on top of a library, which brings the next two problems: Libraries are typically randomized by ASLR and the exact library (libc) is not known. Let's discuss the first problem. With the overflow the function pointer can be overwritten, however, the input was length-limited and spaces were not allowed, but the first argument to the function pointer invocation was under control (the input itself). I therefore changed the function pointer to a printf-call. The result was that printf got called with my input as first argument. Please note: The printf address is not the address of printf inside libc (which I don't know), instead it's from the binary itself where it calls printf somewhere (this address is static in contrast to the libc address because the binary is not PIE). Next, I changed the first characters of my input to exploit the format-string vulnerability which I created myself using the buffer overflow. I'm doing this because a format string vulnerability can leak addresses, especially possible pointers to the libc (return addresses which are stored on the stack can point to the libc code). Also note that the input was length limited (something like just seven characters). It's therefore not possible to dump data with "%x%x%x%x….%x" because this input would be too long. Instead, I used direct parameter access to access the different contents on the stack with inputs like: %1234$x or %5678$x which are luckily exactly seven characters long and print the 1234th (or 5678th) value on the stack. I executed this attack offline on my own system (with a known libc) and iterated over ~2000 offsets/values on the stack. While doing this I identified two different libc addresses which pointed into the middle of libc functions (because they were return addresses). Doing the same attack against the CTF service showed that the return addresses were also there (with slightly different offsets). So, the next thing I did was to change the format string from %1234$x to %1234$s which means that I try to read a string from the return address. This returns some bytes from the code stored at the return address (all bytes until the first null byte is found). Doing this for both return addresses reveals me two code patterns in the target libc and I also know the relative offset between both by subtracting the returned addresses (and taking ASLR into account; and I also know that these addresses must be inside two specific functions). Moreover, I run an nmap scan to identify the target OS. Next, I wrote a python crawler to crawl libc-databases in the internet for these libc values. After identifying the correct libc version and binary I know the exact relative offset from the libc base to the system function (inside the libc), however, I didn't know the libc base address of the remote system because this value was random at every new connection (because of ASLR). The randomness of the base addresses of libraries is not very high on Linux and can therefore easily be bruteforced (just use always the same address and ASLR will bruteforce it itself). After some minutes the system-function was successfully executed with my seven-character command. However, spaces were not allowed in this attack and therefore arguments could not be passed to the command. To bypass this final restriction, I tried different techniques like $IFS or $'\x20' and so on. Finally I found the following payload: {ls,-a} which worked perfectly in the target bash. As you can see, trying to go further in this challenge helped me to learn even

more.

- In the last paragraph we saw that it's often useful to turn one vulnerability (e.g. function pointer overwrite or buffer overflow) into another vulnerability (in this case a format string vulnerability). This step is very often very useful because some other vulnerability classes have better properties for the attacker, like the possibility to leak addresses to bypass ASLR (format string vulnerabilities and especially use-after-free bugs). For example, you can try to overwrite a return address to point to a printf call to leak data or create a format string vulnerability or to a call to free() where you control the data or to some instructions before a free call where a specific address gets freed. Another possibility is to return in front of the function which reads the input. For example: If your input is length-limited to 40 characters, then you don't have enough space for your shellcode. One solution could be to use egghunters, however, if the application was compiled with NX, you can't run shellcode and therefore must do ROP, but 40 bytes is most likely too few for a full ROP chain. What you can do is to overwrite the return address to point again at the fread() function, however, you return exactly after the instruction which sets the argument 40. This way the current value in the register (or on the stack) will be used as length instead, if this is bigger than 40, you can read more bytes and trigger the same bug again with a bigger input buffer. This trick is very common in CTFs (For example in DEFCON CTF). Please note that in the previous paragraph this was not possible because the argument came from argv, moreover, a one-gadget was also not possible.
- Another very common trick for CTFs is the ulimit trick. As already explained, ASLR randomizes the base address of all libraries. If you start a binary two times, the addresses will differ. In the case of a local exploit (privilege escalation), you can use the ulimit trick. Execution of "ulimit -s unlimited" will disable ASLR for the loaded libraries (verify it by calling "ldd ./application" two times; the addresses for libc will be the same).
- Another common trick is that stack cookies and ASLR can be bruteforced in forking-applications. The reason for this is that a fork doesn't re-randomize the application and therefore all addresses will be the same. In such challenges the application forks for every new connection and the input will not be null-terminated. This is required because to successfully bruteforce a stack cookie or return addresses you must bruteforce it byte-by-byte and if the last input byte is not controlled (e.g.: always null because of the null-termination), this is not possible. The general idea is that you open a new connection, just write one byte too much and overwrite this byte with zero. If the application crashes (= no response), you know that the byte was wrong, and you start another connection. In this connection you try a one instead. Then a two, then a three and so on until you receive a response which means you overwrote the old value with exactly the same value. Then you learned the first byte and can continue with the next byte. After getting the stack cookie you can continue to leak the base pointer (to bypass stack ASLR) and the return address (to bypass Position-Independent-Executables).
- It's also good to know that ASLR on Linux randomizes the base addresses of the loaded libraries, but not the relative offsets between libraries. This means that if you know one address, you immediately know all addresses. On Windows the OS libraries are just randomized at boot time and therefore addresses are the same until the next reboot (because of performance reasons).
- You should checkout preeny, a collection of useful preload libraries which can disable / change default functionality (e.g. disable fork or alarm calls). You can load it via LD_PRELOAD or inside GDB with "set environment LD_PRELOAD …" (https://github.com/zardus/preeny)
- When debugging forking applications, you may want to set follow-fork-mode to child inside GDB. In many cases it's better to directly patch the fork by overwriting the assembler instructions with

NOPs (No Operation Instructions) or you can use preeny.

- Let's assume that your application reads input from the arguments, then you can easily pass arbitrary values with python inside GDB (or in command files) with the following command:

```
Run arg1 `python -c 'print "ABC" + "\x12\x22" + "C"*50 '` arg3
```

- If the application reads input from STDIN instead, it gets harder. In simple cases you can pass STDIN values via python with the following syntax in GDB:

```
Run < <(python -c 'print "ABC" + "\x12\x22" + "C"*50 ')
```

- Now let's assume that you want to enter some data via STDIN, then read the output (which may leak some addresses to bypass ASLR) and then you want to send the next input to STDIN (which depends on the leaked data). In this case the above approach would not work because the input is fixed. So, what to do? One solution would be to bind the program from STDIN to the network and write a python script which sends the data to the network port. This can be done with socat:

```
socat TCP-LISTEN:4444,reuseaddr,fork EXEC:"./application"
```

- Inside the python script you establish the connection to port 4444 and directly after that you add a raw_input("wait for debugger") call. This way you have time to attach with GDB after the connection. You can attach with GDB in a second window with:

```
Gdb -x commands -p `pidof application`
```

- This was the old way which I always used, however, nowadays you can also use the PwnTools framework to do exactly the same. When developing the exploit locally, you write the line "p = process(".\application)" in the script, if you want to run the exploit against a network port, you change it to "p = remote(IP,PORT)".
- Another important fact is that you should automate as soon as possible. For example: If your target application supports commands to register a user, login, logout, send messages between the users and so on, you should create python functions for all these commands. If you later read the code in IDA Pro and identify that maybe a bug can occur when you send 1000 messages, you should already have the possibility to quickly verify your assumption by just looping 1000 times over the send_message function in the python code with two lines of code. This flexibility allows you to quickly check different corner cases.

So far, I only explained ways to reverse engineer and analyze binaries, however, I didn't talk about how to make this as fast as possible. For beginners I recommend to just start and understand all the functions before you start to hunt for bugs. However, as experienced player you may want to identify bugs as fast as possible (in attack-defense) CTFs. Here are some tips:

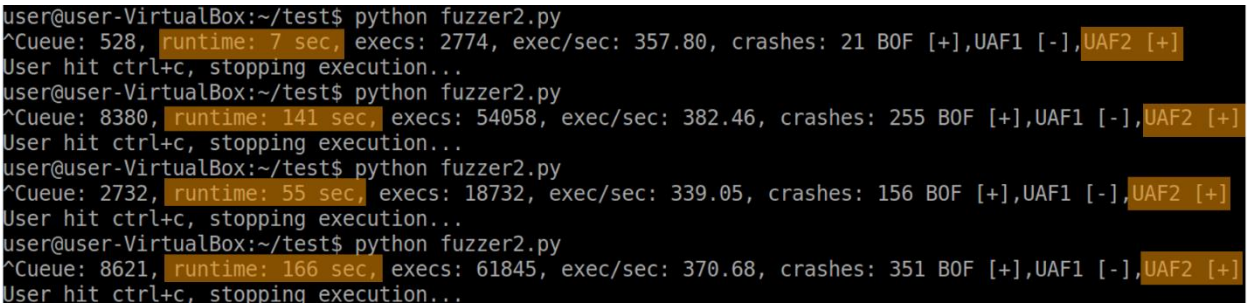
- A good idea is to directly decompile the full binary with IDA Pro (professional license) and run a static analysis tool over the code. Before you decompile it, you should run an IDA Pro script which automatically creates arrays (so that buffer sizes are correct in the C code). This way you can quickly identify lots of bugs.
- Another thing which I do is to check common functions which lead to security vulnerabilities. For example: I go to every call of printf (IDA Pro tab functions, then double click the function and press x for the cross-references) and check if the first argument is user controlled. I'm also doing the same check at runtime with a printf-wrapper with LD_PRELOAD. At runtime you can check if the

first argument to printf (and similar functions) point to a writeable location and if so, it's most likely user input and can therefore lead to a format-string vulnerability. Other functions which I check are memcpy, strcpy, and so on. Moreover, I check every call to "free" by decompiling the associated code and checking if the variable which was freed is set to zero afterwards (if it's a local variable and not used anymore in the function it's also ok). If it is not set to zero it can lead to a use-after-free bug.

- Another good idea is to run a fuzzer against the binary as soon as you get it. There are two types of binaries which require different fuzzers: Binaries which read binary formats and binaries which have an interactive prompt. If the application reads a binary format (e.g. .jpg files, .cfg file, .wav, …) you should use AFL (American fuzzy lop http://lcamtuf.coredump.cx/afl/) to fuzz it. AFL supports fuzzing closed-source binaries with the qemu-mode (-Q flag). However, in CTF applications with interactive prompts are way more common and therefore AFL is often a bad choice. Instead, you should use a fuzzer which mutates the order and number of commands and arguments. The best choice for this is radamsa (https://github.com/aoh/radamsa), a tool which can mutate your inputs.
- Since radamsa requires valid inputs to mutate, it's best to directly generate them while playing around with the binary. For example, the first steps which I do are running the "file", "checksec" and my auto-analysis commands. After that I start to play around with the binary. I start the binary with the following command:

```
tee `mktemp inputs/input.XXXXXXXXXXXXXXXXX`  | ./application
```

- This way I can normally interact with the binary, but all my inputs are stored in input files. For example, I can later re-execute the same application flow by checking for the latest timestamp in the inputs directory and then just replay it with "cat file | ./application" (if the application doesn't use random variables; if it's using random variables I use preeny to seed it with a static value). Moreover, it has the benefit that I have automatically created valid input files for fuzzing with radamsa and it very likely triggers all interesting paths because I manually check every possible interesting path when playing around with the binary. After that I just run a simple 100 Lines-of-Code python script which calls radamsa on the inputs and after that the application with the mutated inputs. For example: The SECCON chat binary contained three vulnerabilities, two use-after-free bugs and one buffer overflow. AFL can easily find the buffer overflow and the simple use-after-free bug. However, it will not find the difficult use-after-free bug. This is not because AFL is bad (quite the reverse: AFL is extremely good and well designed), it's because AFL was not developed for such applications. With the custom fuzzer I found all three bugs in under three minutes:



In the image above you can see that this 100 LoC script found the BOF (Buffer Overflow) and the

2nd UAF (Use-After-Free) bug every time in under three minutes (most of the time in under one minute; Side note: The UAF1 was not found in the above image because I patched UAF1 in the binary). To say it one final time: If the application is interactive and you can send commands to it, fuzz it with radamsa (like the chat binary). If the application reads binary data, you should fuzz it with AFL qemu mode. Example: The iCTF pokemon challenge allowed to save the current state of the game and reload it later. Because this is "binary data" (not interactive), this should be fuzzed with AFL which reveals the vulnerability in exactly one second!

- If you use AFL for fuzzing, you should use wordlists.
- Independent of the fuzzer you are using, you should use a heap library for better bug detection. Some vulnerabilities do not lead to crashes and therefore you maybe find a bug but don't notice it! The SECCON chat binary is again a good example because all three bugs do not crash with the default heap implementation. Changing the heap implementation is very simple – AFL ships with libdislocator which is a custom heap implementation. You just need to compile it with the "make" command and after that you can use LD_PRELOAD to change the heap implementation:

```
user@user-VirtualBox:~/test$ LD_PRELOAD=/home/user/test/libdislocator.so ./chat
Simple Chat Service
1 : Sign Up     2 : Sign In
0 : Exit
menu > 1
name > a
Success!
menu > 2
name > a
Hello, a!
Service Menu
1 : Show TimeLine       2 : Show DM     3 : Show UsersList
4 : Send PublicMessage  5 : Send DirectMessage
6 : Remove PublicMessage                7 : Change UserName
0 : Sign Out
menu >> 7
name >> abc
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

In the image above you can see that the application crashed. The reason is a buffer overflow because using the command 1 ("sign up") I created a user with a username length of 1. Later I used the command 7 ("change username") to change the name to a longer one. Since we just allocated a buffer for a length-1 username and now write 3 bytes ("abc") into it, a buffer overflow occurs. If you try the same inputs with the normal application (without LD_PRELOAD libdislocator), the application won't crash. You should therefore always use a heap library when testing and fuzzing the application! On Windows you can use GFlags to change to the page heap and you can also try DrMemory from DynamoRio.

- Instruction counting can be an interesting "side-channel" to quickly solve reverse engineering tasks. For example: Assume the binary asks you for a password but the checking code is heavily obfuscated and encrypted. Of course, you can start to de-obfuscate and decrypt it, however, a much simpler approach is the following:

```
$ for i in {A..z}; do valgrind --tool=callgrind --log-file="output.txt" ./main `echo $i`1
2; cat output.txt | grep refs | awk {'print $4'} | tr -d "\n"; echo " for >$i<";done | gr
ep for
117,453 for >A<
117,453 for >B<
117,451 for >C<
117,453 for >D<
117,453 for >E<
117,454 for >F<
117,453 for >G<
117,453 for >H<
117,453 for >I<
117,453 for >J<
117,453 for >K<
117,455 for >L<
117,453 for >M<
117,453 for >N<
117,453 for >O<
117,460 for >P<
117,451 for >Q<
117,453 for >R<
117,453 for >S<
```

You can see here that I iterate though all characters (A…z) and I call the target binary (main) passing the currently checked character as argument. After that I grep the number of executed instructions (which I measure with the callgrind tool from valgrind; PIN or DynamoRio can do the same). For the input "P" the number of executed instructions is higher and therefore "P" is most likely the valid first character. So, we know the first character and can now "bruteforce" the next character:

```
$ for i in {m..z}; do valgrind --tool=callgrind --log-file="output.txt" ./main `echo P$i`
1; cat output.txt | grep refs | awk {'print $4'} | tr -d "\n"; echo " for >$i<";done | gr
ep for
117,460 for >m<
117,460 for >n<
117,460 for >o<
117,460 for >p<
117,460 for >q<
117,460 for >r<
117,460 for >s<
117,460 for >t<
117,460 for >u<
117,460 for >v<
117,467 for >w<
117,460 for >x<
117,460 for >y<
117,460 for >z<
```

The above command is completely the same except that the "P" was added as first character to the argument and from the output we see that "w" is the second character of the password. You can continue the same steps for every character of the password. I already solved several challenges using this approach. Please note: You can also solve this type of checks with a symbolic execution engine like angr.

This guideline was written by René Freingruber (@ReneFreingruber) on behalf of SEC Consult Vulnerability Lab.