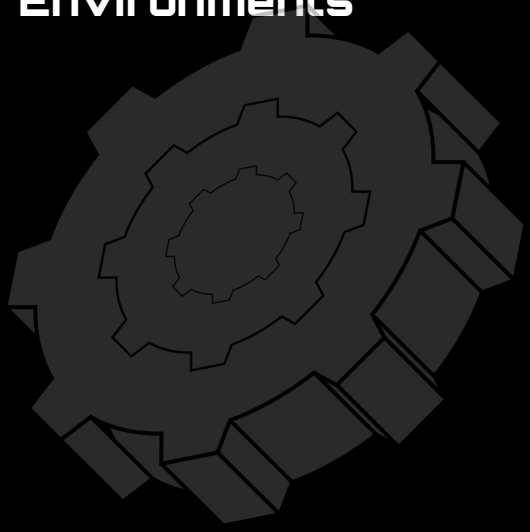
[Free Release]

SkyPenguinLabs

Proudly Presents:

REC8 - Reverse Engineering & Pattern Recognition Training

- Pattern recognition? What?
- Tired of just reading strings?
  - Learn how to use different info to your advantage
- Training Yourself for Newer Environments



Simplicity > Complexity

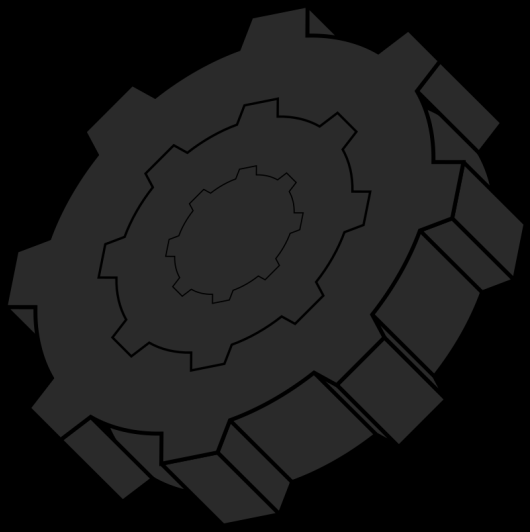
Author: @Totally_Not_A_Haxxer

///.//./.// >>/ VISIT SKYPENGUINLABS.WTF TO SHOW SUPPORT REC8

TABLE OF CONTENTS

FOR THE COURSE: REC8 - Reverse Engineering & Pattern Recognition Training

- **Section 0xFF** | *Conclusion & Final Notes*
    - Concluding this course - Reiterating over important information. Additional final notes and information to take with you!
    - A personal thank-you for your participation in this course and continued support of Sky Penguin Labs course development!
    - A few recommendations for courses to check out next!



SkyPenguinLabs_REC8

SECTION: Section 0x00 | *Course Synopsis & Details*

COURSE: REC8

Pages: 36

Difficulty: (x) Beginner [o] Intermediate [o] Expert (Selected)

Synopsis:

This course was designed to walk you through what pattern recognition actually means, and how as reverse engineers, we essentially use it on a daily basis *(And yes, pattern recognition as - the thing you have probably heard a million times on tiktok or scrolling instagram reels wondering what went wrong with the world).* We will then leverage the recognition and understanding we have about it, and discuss some methods we can use to train, and extend our recognition skills!

We won't get deep into every single method, but we will be covering a variety of roadblocks that are common in today's world, such as compiler optimization, or compiler data standards such as the notorious **I.E.E.E 754**. Alongside understanding the result compilers spit out, we will also be going behind the scenes to see how code gets transformed into binary, which gets transformed and translated by our decompilers. We explore this by discussing various methods such as:

- **Building a custom application that is relative to your needs**, and then reverse engineering it
- **Taking from case studies to learn** about targets or individual sets of vulnerabilities
- **Exploring system libraries like Windows DLLs or Linux .so files to reverse engineer and identify how system API functions appear** in compiled binaries after being statically linked and optimized away, essentially reconstructing what those APIs would have looked like if they weren't inlined or stripped during compilation

With that being said, we wish you the best of luck! The next section covers the introduction for this lesson!

SkyPenguinLabs_REC8

SECTION: Section 0x01 | *Introduction & Authors Notes*

As reverse engineers, it is practically our jobs to dump a bunch of code onto a screen from a disassembler, and map out specific fragments that catch our eyes. It may not even be code, as in the most brutal cases, it can just be text based data that has absolutely no meaning to you right away. Whatever it may be, pattern recognition can both aid you and be the reason you have a few pitfalls when trying to execute such tasks…that is if you do not understand its boundaries, not understanding where relying on it can mislead you, or even how to train it…

Though, despite the pitfalls, which we will explore more in-depth, the exciting part about pattern recognition is that it lets us build a mental library of patterns we can recognize and rely on in real-world reverse engineering scenarios. While we won't dive too deep into hands-on work just yet, we'll walk through key examples that take us from a general understanding of pattern recognition, to developing and extending that skillset, and finally to applying it effectively in practice.

Before we start, I figured it would be important that we start off with some general notes.

[Q] - Why is this lesson not super practical?

- While we want to release free lessons that are enticing, we can only do so much! This lesson would have been huge if we stuffed everything we wanted to into it, so we decided to make this free version, and potentially invest the time in creating a second version if enough people take interest in this one

[Q] - Do you like the new format?

- We have been working really hard to push out some cool themes and finding ways to make the watermarking more unique, specific and non-annoying while still effective (*watermarks have to kind of be annoying though*). I want to also acknowledge the anonymous designer behind the articles ^_^ Thank you so much for all the hard work put into the format/layout behind the documents. We spent a large amount of calls brainstorming over these LOL!

Finally, I hope you enjoy it, if you want to check anything else out- don't forget to see us **here**.

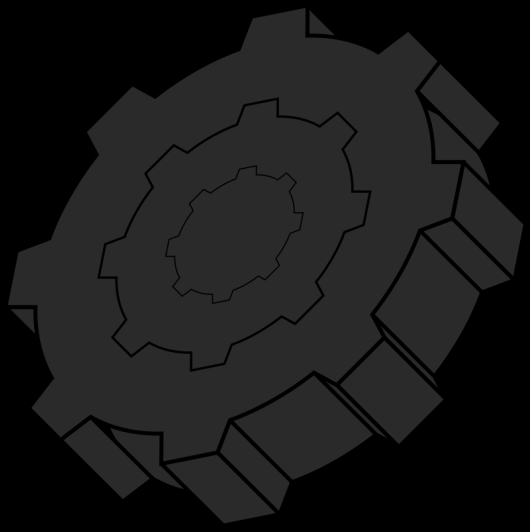
SkyPenguinLabs_REC8

SECTION: Section 0x02 | *Prerequisites for This Course*

This course is purely theoretical. There are some parts where we do go a little bit practical, but it is not required for you to follow along. The practical examples are just here to back the theory up into what the discussions are about.

If you want to follow along, know that I will still be discussing exactly what I am doing to get to where I am. Additionally, for all readers, we will be using IDA-Pro for some of the reference samples. The applications we will be using for demos will be published on **this** repository and will be also linked all throughout this lesson.

Finally, this is a pretty easy lesson to understand, it is literally the core fundamentals of the software reverse engineering world most people start with: static analysis. We just expand the way we are analyzing the code and talk about how you can train yourself to essentially be better at picking out discrepancies amongst data being displayed in-front of you!



SkyPenguinLabs_REC8

SECTION: Section 0x03 | *Pattern Recognition & RE - WTF Is it?*

Pattern-recognition very much seems like something floating around here and there that kind of became a buzzword over the last few words - pattern recognition. I am sure you yourself may have heard of it. If not, well- then count this the perfect section for you.

Directly speaking, in relation/context to human psychology, **pattern recognition** is simply just a cognitive functionality and or **skill** that is built inside of our brain naturally. Because it is a skill, it can be trained! But of course, we have to understand exactly what or how pattern recognition works in our brain!

Fundamentally, how does it work?

When it comes to pattern recognition, at the very bare bone level, there is a lot going on! It is the brain afterall. And since we are not trying to be neuroscientists here, we won't try to act like we are one and say we have the god spoken answer and spit it out. But we will break down some fundamentals.

Generally speaking, pattern recognition simply involves your brain taking information in, processing it, and specifically looking for unique patterns where the pattern is defined as a consistent, predictable sequence of information (*like sights, sounds, or movements*) that the brain recognizes and matches to stored memories to make sense of the world, learn, and predict outcomes. This is pattern recognition.

Pattern recognition stems from this baseline terminology and splits into multiple different categories which are the following:

- **Visual** - Typically the most common, used to recognize faces and patterns in text such as those on a book, and more. This is more so helping us crunch those docs down…haha
- **Auditory** - This involves the ability to identify the patterns in sound inputs mainly in relation to languages or detecting changes in volume or variables in the audio itself. We would not be using this much, but it has lent its hand in a variety of research around audio devices- so, there is a place for everything we suppose.

- **Temporal** - Everything prediction in relation to recognizing patterns over a set amount of time.

- **Mathematical** - A skill that involves being able to identify patterns in numbers, sequences or shapes. While it seems like this one should not really have its own fields, given the other 3 above, this one is the most used by us as reverse engineers. Mathematical pattern recognition is a distinct neurological function because it involves an abstract system of magnitude and quantity that is only partially dependent on visual input. While mathematical symbols are visual, the patterns themselves, the concepts of quantity, logic, and abstract relationships, are not.

Paying attention to the category, as you could have guessed by the meanings, is pretty important if you are looking to expand your pattern recognition skillset so you can actually use it more.

Let's take a look at a small example.

If you were a soccer player, you would not try to expand your temporal pattern recognition, instead, you would most likely lean into visual pattern recognition because you are trying to act in the moment on a field, playing a sport, running around with a lot of **mental stimulation** going on.

But what about people like us?

How do reverse engineers use pattern recognition as a skill?

As I mentioned in the bullet point above, when discussing mathematical pattern recognition, I mentioned this was something you use frequently as a reverse engineer. This is most *__likely__* true based on the definitions above.

- **[B] -** This is a biased note, because the owner of this claim (Totally_Not_A_Haxxer) only has physical, personal experience in grasping this understanding, but no statistics specifically for 'reverse engineering' and 'mathematical pattern recognition' that alludes to a form of support for the above.

Since we don't have much stats and proof, let's think about this for a second, and think broadly.

How many cases of reverse engineering do you think **probably** require some form of math? I will give you a small hint to that number that can help us get an

estimate…if in 2023, **more than 100 million different strains of malware were discovered** or named/identified, how much of that most likely used some form of binary obfuscation? Additionally, how many had to calculate a payload's positioning in memory?

If you know anything about obfuscation, you know that even when it is not trying to be the most complex process, there is a LOT of math involved. Here is a hand obfuscated example **here** from the IOCCC competition! As you can imagine, being automated, and stuffed inside of a binary, it's no better.

That being said, all the people that sat down to reverse engineer that program to identify the threat, just in software along most likely had to recognize some mathematical sequences, and even when breaking down functions and patterns of assembly, the need for understanding how to pick out the identifiers and symbolistic meanings is pretty important- especially in threat scenarios where information is extremely time critical.

- **Fun Side Note -** A shot in the dark for those that are new. But if you have ever watched a CTI (*Cyber Threat Intelligence*) analyst actively work in the field, or ever gotten the honor to help one, or even communicate with one- have you ever been able to ask them what their most time critical scenarios were? I bet you they have one. Maybe not one they can share, but regardless, time is a serious factor that many people when educating do not really reference in the RE world, which those such as CTI analysts, literally depend on for life or death in the field. This is largely due to the fact that time-critical decision-making is a situational awareness skill that can only be fully developed through real-world operational experience; many educators are not fully experienced, and instead start their careers having more educational backgrounds, rather than work backgrounds. For CTI analysts, time is a mission-critical factor, intelligence must be rapidly given or informed to stakeholders such as the DoD (*DoW as of 2025*), federal agencies, and trusted partners engaged in operations to enable proper defensive action at the perfect time. Delays as a result of reverse engineers wasting time, or not being able to obtain information due to roadblocks (*personal, or not*) can lead to widespread impact. If we want to be all Hollywood hackerman over here, IRL - compromise of national infrastructure, loss of data for almost every provider affected, and downstream civilian-sector disruptions as a result of simply wasting time gathering intel is extremely prominent. This is why, **when a quarter of the entire world goes down**, it's a pretty huge deal. Especially when it's one of the world's largest AV protectors. Even devs have to be this fast! Can you believe that?

By this time, you can probably figure we are not talking about the pattern recognition you may be thinking of, which is simply the programmatic version of what we are talking about.

Pattern recognition outside of YOU!

Since this is a free reverse engineering article, we can not hesitate to break down where pattern recognition lies inside of reverse engineering but outside of you. Meaning we are breaking down exactly what it means to take this skill and apply it to other areas.

- Context Note: Programmatically is a way of saying or expressing a concept that has been implemented into a program via a programming/scripting language.

When reverse engineers use pattern recognition to their advantage using programs, usually it is built out to:

- **Parse symbolistic text** - Usually, there is a set amount of input data given to the program, usually in a specific data format that the program can understand and decode such as **hexadecimal**, **unicode**, plain binary data, or whatever it may be. Then the program will decode this text, and look for specific patterns using a variety of programmatic pattern recognition techniques. This involves filtering through plain data, working to make sense of it by using regular expression algorithms, or using various searching algorithms if the content resembles a specific path that needs to be taken. This can also be memory from a program that gets **dumped in ASCII**. Symbolistic matching is derived from parsing symbolistic text and attempting to make meaning, but differs slight;y.

- **Symbolistic Pattern Matching** - In many cases, developers or reverse engineers can take an input series of data and make a sort of pattern that a program can test the input against. Instead of checking pre-defined conditions, scanners that typically accept patterns will be completely dynamic in their nature. Which means whatever pattern you throw at them, they should be able to find it. Most beginner devs experience this with **Regex**. However, in more advanced scenarios, patterns are built into a body of 'rules'. Where a 'rule' is a set of pre-defined logic for the parser to match, not only including the symbol, but individual characteristics the text MUST have to ensure it matches. This is often seen as a pass/fail form of testing. Think of YARA.

- **Entropy Measurement -** Entropy analysis is a technique often used in reverse engineering to identify sections of a binary that are encrypted, compressed, or packed. The process involves dividing the file into fixed-size chunks and **calculating the entropy for each**. These entropy values can then be visualized using a heatmap or a graph, where high entropy regions (*often indicative of encryption or compression*) appear much more distinct from low entropy regions (*which may contain code or readable data*). This visualization helps us quickly identify deviations in regions of that file which may require further investigation or are relevant to a specific hypothesis, for example, locating a packed section that contains the main payload of malware.

A common problem people often have with "starting out" is that they rely too often on frameworks, or plugins which already automate the task of pattern recognition in reverse engineering to tell them whether or not something is worth looking at - e.g: auto detecting plaintext credentials. But believe it or not, in order to work with some of these methods, you yourself need to exercise your own pattern recognition skillsets. In fact these tools could not be even built if humans were not already absolute machines at doing this.

Another problem on the opposite side of that, is specifically related to a misunderstanding of how exactly to train this skill which results in a **proper extension.** (*you can train wrong, but still get a good result which makes you think its been extended properly when it truly has not*)

Truthfully, a lot of people also with this, expect that if they 'try' hard enough at it, that they will feel it, but you won't. Believe it or not, this is one of many reasons imposture syndrome hits you HARD - Not recognizing patterns of growth or success can fuel imposter syndrome like a mf because it prevents your brain from building a consistent, internal belief in the skills you learn or have picked up prior to execution.

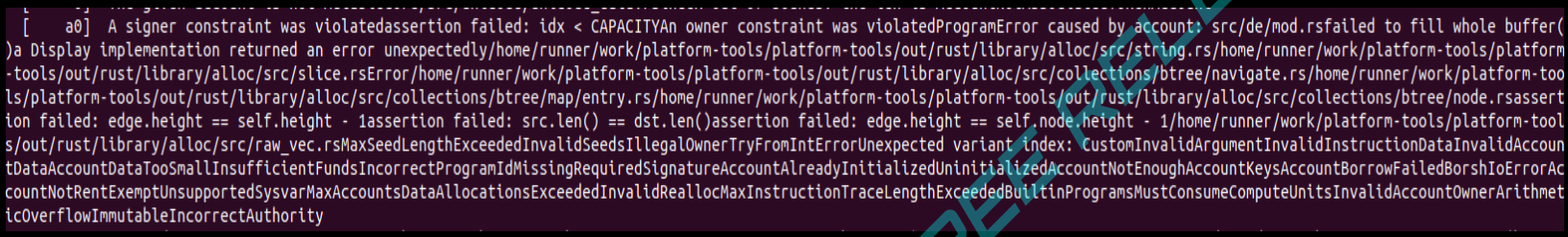
Pattern recognition, while having the ability to show off its progress and have its progress be felt, is almost impossible to observe in the direct moment that it's being grown or executed. This is also due to not just the natural inability to observe subconscious processes, but also due to the fact that you most likely try so hard on trying to train yourself, you end up not training at all and are instead distracted.

So how do you train something you can't really feel? Let alone, how do you even know it exists?

Well, similar to how you walk, act, talk, and more, you eventually get into the habit of it, especially when it is literally a part of you.

Here, let's explore a pretty cool sample. One thing I commonly do when I show screenshots is take advantage of drawing a LOT to pinpoint things. You'll notice this in some courses. But have you tried looking at some of the shots without it? Let's check one out.

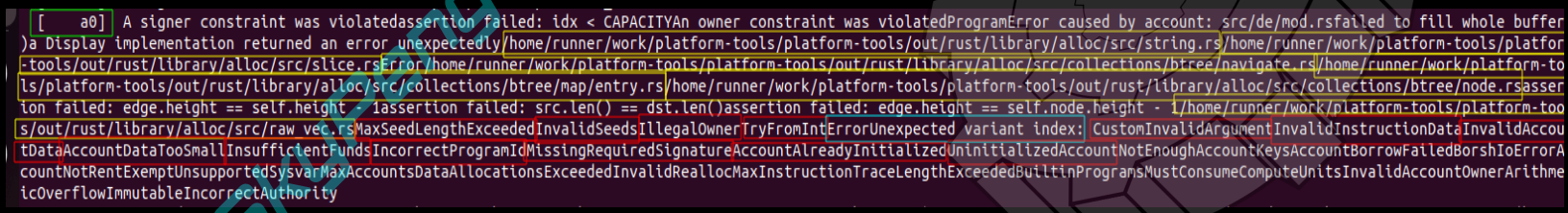
A little bit simpler, but can you pick out the patterns in this basic image had you just slapped this on a desk?



If you were to seriously sit down and look at this image, would you have:

- Noticed the category of data?
- What does all the data share in common?
- How is the data structured and how persistent is each deviation?
- What technology did this come from?
- What compiler was this output likely from? What output even was this?

Immediately, pattern recognition for untrained eyes does not see much, and may take a tiny bit longer to map out all of this information. However, what if we….did this-



Now just looking at this your eyes probably instantly dragged into the spot where everything was highlighted. As you can imagine, this is natural to most people - look where the highlights are.

But if you were to actually see and understand what the highlights mean…for example,

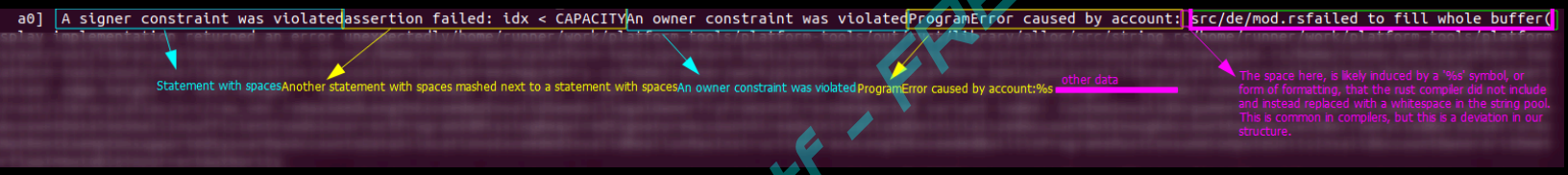
- Yellow gives us information that this is most likely the string pool of a rust compiled binary. The presence of **.rs**

- Red gives us detailed information about what this section of strings most likely resembles - errors. This is because all of these names are names of common error schemes you can come up with yourself and or have heard. Such as *'Insufficient Funds'*

- Green - verifies we are looking at a string table (*likely*) from an output column similar to readelf or objdump or really any binary dumping utility.

- Aqua - Resembles an indication of separation in structure.

Aqua is where things get wonky so let's get rid of the mess and try to find another approach to looking at this information.



As you can tell, we start to pick out different characteristics, each part we start to explore and isolate seems to lead to some new format.

In case you are NOT getting it: In this screenshot, we are referencing a string pool from a rust program. This string pool, because we isolated a lot of the noise in the output, and were able to focus on a specific row, were able to identify that specific data is usually adjacent or in some shape or form related to one another. In this case, we have a set of error messages:

['a signer constraint was violated', 'assertion failed: idx < CAPACITY', 'an owner constraint was violated',]

But then we come across something interesting in the pattern. We notice, there is a spacing after the error *'ProgramError caused by account: '* then we see another new error message which we call new, because it starts with a capital letter, indicating a new sentence or phrase, as we also identified that the first letter and last letter of the right and left error messages in this case were pushed together in the string pool, but would still contain word breaks used if there were some in the phrase itself.

So what can we imagine would cause a space?

Since this is an error, and it seems to be referencing ‘account’ in relation to a program, we can infer that the context of this statement is to format some account identifier to the error message. In this case, the error is being formatted with a format identifier, which explains the whitespace to some degree. Though, keep in mind this is an inference on the data.

Because of this exercise, I can almost guarantee you the next time you look at a string pool in a program, or for that matter, a cluster of random strings that seemingly put together, you may start to easily pick out patterns about the way data is structured in that pool just based on what we walked through.

Now based on this exercise as well, think back to what we were talking about before - automation.

How could you see, now that you grasp a generalized structure of the data resembles in the screenshots, *this being automated to be scanned?*

It may not be the best, but maybe we could:

- Differentiate error messages by a set length, character expectation (*since error messages have coding standards in rust, such as this common format here*)
- Extract file paths by pattern matching all file paths and specifically targeting .rs files by checking for only that extension
- Sort through data by assertion could be done here as well via pattern matching, assets have a very clear format here.

In which case, pattern recognition now goes out of your hands and into programs. Of course, the implementation that transitions from the thought process of what pattern recognition actually is into a program is a lot different but is actually not that hard once you grasp the core route of what it is: sorting through data. That is at the very surface level.

Now that we can say we

- *Grasp what pattern recognition is*
- *Grasp how pattern recognition works a little bit inside and out of ourselves*
- *And grasp the fundamentals of training, we can get into exactly that, training!*

SkyPenguinLabs_REC8

SECTION: Section 0x04 | *Training Methodology - Get, Freaking, Building!!!*

Training is a pretty important thing to get right, and in scenarios like this, it can help you get around so many different roadblocks in the real world of reverse engineering, that when you get there, it feels like a boulder to lift out of the way! A huge problem with people that start out in cyber is that they often do not know where or how to **source** information. This is part of the reason, cybersecurity is not a beginner field (*and yes, cyber is referring to cybersecurity in this lesson*). And even when they do, (*i.e: going to instagram, youtube, tiktok, etc*) they usually end up in the wrong places, discouraged, misled, and in the worst cases, not even caring to double check the information they are receiving. In other words, a lack of discernment is present.

Training, and learning how to source training, especially without paying thousands of dollars out of pocket is not easy either. As we will explore in another free article, sourcing information is incredibly important for researchers even beyond the learning, and if you learn without knowing how to source, you will find yourself even more stuck in roadblocks that are even more detrimental.

Though, on the other hand of this, one of the best ways to learn the sourcing of information is experimentation - through experimentation, you gain experience building things which will most likely fail, forcing you to Google something in relation to it, likely dragging you down a rabbit hole if you are truly interested and not GPT vibe coding your way through the world.

What better way to train yourself, than to experiment with new flaws by learning how to build your own training labs for those flaws?

If you are not new around here, especially in the RE world with some of the stuff I talk about, you may have heard me already reference that building things is pretty important.

Though I have also argued that understanding programming is also not necessary for every single reverse engineer in the world. While this is true and extremely counterintuitive (*so it *seems*), I usually followed that up with: but, you won't be the best, and you probably won't even be able to maximize the potential.

This is also true.

Building your own cyber-range / training

Building your own labs, which basically involves taking something like C++, building a basic GUI app inside of it and stuffing it with flaws is the most simplistic yet best way to pick up on building your skillset all the way up to, say learning how to exploit a critical flaw or bug.

Now, let's get this solidified. What exactly do I mean? How can I learn something If I implemented the flaw and then hacked it? Would that not defeat the whole purpose of a CTF? Would that not fail to deliver a wide range of skillsets that's widely applicable?

Well, it's as if that's the point. It's actually NOT a capture the flag, and the goal is never intended for it to be a form of CTF. This is known as a cyber range! CTFs, especially professional ones held by Google or IOCCC are designed specifically as a form of competitive competition. Not as a form of training such as cyber ranges are.

Building a cyber range involves taking an in-depth understanding of a real world concept, and bringing it to life by taking examples of said concept being executed/utilized in a real world scenario, and building applications around that to simulate real world environments.

Sometimes, cyber-ranges are extremely annoying to build because it can require a lot of patience in programming, and a lot of research. But this is exactly the point.

Many people get this thought that **training** on CTFs is what you should be doing, but you should instead be training on cyber-ranges, but be testing the skillsets you learn from that training on a CTF which allows you to collaborate a multitude of the skills learned entirely throughout the development or testing of the cyber-range. A common cyber range is WebGoat, a deliberately built insecure application that barely touches the line of what a cyber range can be or is.

On the other hand, why develop one if they already exist? Why am I doing all the work?

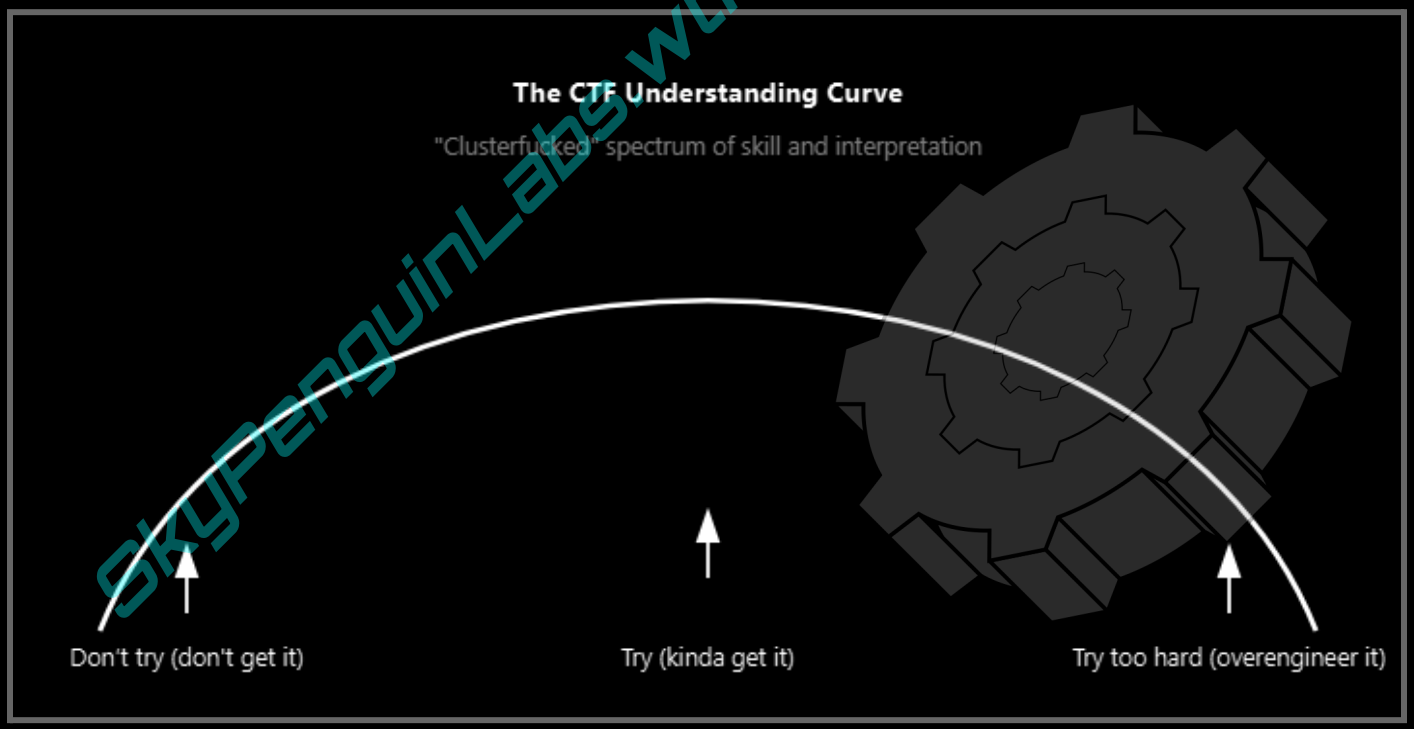
Truthfully speaking, this opens up a much larger and wider learning experience where you internalize both how developers accidentally introduce bugs and how attackers later exploit them. Especially when you are hands on - both sides.

So to conclude my point: unlike Capture The Flag (CTF) competitions, which are gamified and time-pressured environments primarily aimed at testing skills through intentionally designed challenges, of which, some may be unrealistic, overly designed or the opposite, under developed and under thought; cyber ranges usually have a goal to replicate production-like systems and attack surfaces, creating more realism needed for more concise results & predicable outcomes in offensive and defensive operations.

Sometimes, this involves setting up real infrastructure - such as Kubernetes pods or AWS endpoints.

I'd like to note that the CTF landscape is clusterfucked (*chaotic / fragmented*). This is largely due to widespread misconceptions or misunderstandings about what a CTF actually entails, if they should even be standardized, and etc. As a result, participants fall across a broad spectrum: some have a vague idea and give it a shot without fully grasping the objectives; others avoid participating altogether due to confusion or intimidation; and then there are those who dive in deeply, creating highly complex and technically sophisticated challenges that can be difficult for the average participant to engage with.

I drew up my meme view of this personally: it's a curve.



Of course this is not meant to be practical. And I do not seriously mean to be rude, but genuinely, this is the way it feels with most CTF providers out there. And there are many people in the industry I have talked to, and out including those that I help all

of which give similar outlooks to the current state of the CTF world. There are providers out there working on a change, but- its going to take some time.

No doubt, there is definitely some really cool stuff out there to do, but on the surface level, it is either too beginner and too simple which is unfortunately where most people that want to practice or learn new skills end up getting trapped in.

I must add though: even for beginners, despite building your own range being a little difficult, or even building an app towards a specific flaw you want to study needing to take a lot of time towards R&D, the learning experience is so well worth it in the end, especially if you get the same if not more experience than slapping down thousands for useless courses.

Building your own training - Good to know before diving head first...

Creating a personal training environment isn't always straightforward, so I wanted to begin with a simple example to show how the process works.

A key scenario might be exploring the different approaches to SQL injection. To do this, you could design an application that relies entirely on SQL, either in a simulated production setup or in an environment that closely mirrors the one you normally work with.

But starting with this is not always easy, even if you are a beginner and looking at something as simple as above. Why? Well, we come back to the same problem before that haunts learning - sourcing information.

The only thing with building your labs is? It actually gets you to learn how to source and verify information.

See how all of this is tying in again?

When you build and design your own lab environment and dedicate time to studying the material needed to understand how it can be exploited or how it even gets built in the first place, you'll have to verify the accuracy of the information you gather about the vulnerability type or flaw you're focusing on and the app you are building it in to train. Practicing discernment here is critical, otherwise, you risk introducing bigger issues or inaccuracies in your own training data (*pun intended, wink wink*)

Plus, you will most likely face errors in your results which force you to go back and fix them in code before you can finish your goal of *'completing the training via exploiting everything'* (*typically the end goal or to retrieve a specific subset of*

information in a timeline). Because CTFs allow you to just play through, hardly any research is needed to comprehend how the flaw is executed on the backend, additionally, with writeups providing direct answers, there is almost 0 R&D efforts needed to surface, 'beginner' level CTFs, especially when AI/LLMs are added into the equation for solving questions during competitions.

As CTFs are clearly built specifically for competitions, they are there to help you exercise what you learned **in training**, they are **not there** to **be the training**. When you build the training and go through the training yourself, you find yourself in many rabbit holes, both on the defense and offense side that take more than just all R&D, a little programming + some logic to make work.

For some people this may be seen as a lot of work, for others, this may be rather simple. I suppose, it depends on the person's point of view, but ultimately, training is always going to be more useful for building skills than testing them out (*especially if it's your own lab*)

Building your own training - This is not to say CTFs are bad

Over time, I have expressed this formal opinion to many people, in which I immediately got shit back for someone being offended because of such words. Often, this is a result of people being self-conflicted about what other people think about their creation. Such as their CTF being criticized.

I must assert that developing your own training environment is something that is not going to replace CTFs, because like CTF creators trying to make them 'training' it does not work. **CTFs are meant to be CTFs**, training is meant to be training.

This is NOT to say that CTFs can NOT be educational, because they can be and most times are intended to be or teach you different things about a specific case scenario, setup, or system. So there very well are subsets of CTFs for beginners who want to learn or become educated about such topics like: CAN replay over an automotive ECU, buffer overflows on a remote websocket connected to some large infrastructure, and etc.

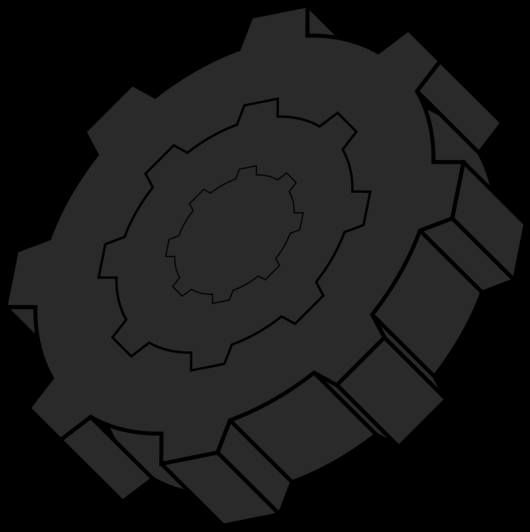
And with all due respect there are some MASSIVE CTFs out there that can teach you SO much about a new space. But there is a reason why in certifications and other 'training' you must submit a detailed penetration test report that includes screenshots of "proof flags" which are still not flags. The hands-on exam for OSCP for example is not a standard Capture-the-Flag (CTF) competition but a realistic simulation of a penetration test, and the course curriculum (*the training before the simulation*) is meant to prepare you for that. CTFs help you practice before the simulation.

Training typically refers to structured, goal-oriented education, often tied to certifications or professional skill development, designed to guide you through a series of tasks that may include identifying vulnerabilities, analyzing them, and documenting your findings in detailed reports within a simulated environment.

A conclusion this time!

To finish this section off, building your own training environment is worth it. While it will not replace the fun and niche characteristics that CTFs can bring to the hacking world, it will help you crunch down a 488 page standard that came out in 2020 (*RIP engineers*) and make sure you can do your job enough.

Additionally, don't only focus on training either, as CTFs can be a useful aid to prevent burnout, and because there is LITERALLY MILLIONS of CTFs out there, you are bound to never run out of brainrot ideas to come up with that are different to implement, or fun to hack and work with. I think both avenues are just as important, but if skill is a priority, training environments are where it's at!



SkyPenguinLabs_REC8

SECTION: Section 0x04 | *Training Methodology - Working in unfamiliar environments*

Since we already have something in this article that is widely theoretical, I also wanted to throw something practical in here.

Upon this free release, and release of this new lesson format we are doing here, we have one new article coming out that is actually really cool! It talks to you about taking a blackboxed golang plugin, and being able to reverse engineer it all the way to being able to use it in your own program in a dynamic state! This was a super fun, 50 page lesson that goes deep! In this lesson, it primarily walks you through navigating an entirely new environment!

The one thing we talked about in that lesson was how absolutely important it is for reverse engineers to be fluid (*the constant ability to adapt to change*) if they are doing constant research or are cycling through contracts. This is because of how widely different individual environments can be if you are not specifically in a designated area of RE - for example: malware analysts will have to not just know how to work with binaries compiled with C++, but ElectronJS apps that are packaged, minified, and obfuscated, or custom bytecode and even in the case of TDL3, custom network protocols and more! It's wild!. (*Check here for the edited case study on TDL3, which references on page 10, the custom protocol used by the C2 for TDL3*)

Staying fluid is the only way to stay as fluent as a reverse engineer can be, and one that stays consistent as well. But doing this, often comes with annoying roadblocks that make 0 sense when you come across them.

Let's take an example: you are a reverse engineer with 5 years of experience specifically working on the x64 architecture. You specialize in reversing applications compiled by G++ and VC++. One day, you get handed a weird task, it's SRE related, but it's for an application compiled with the crystal programming language which compiles to native code using LLVM as a backend. In this scenario, you are the only person on hand who has the technical ability to attempt such a task.

- **Unfortunately**: In the real world, whether you are working in a SOC, on red/blue teams, in research labs, or as a contractor, it's common to be assigned tasks outside your current skill set. This often happens when companies take on contracts involving technologies, tools, or threat scenarios they aren't fully

staffed for. This can be for a random clusterfuck of reasons that make 0 sense or something that does. Regardless, as a result, you may be expected to quickly learn new skills, work with unfamiliar systems, or take on roles outside your core expertise (*e.g., a blue team member helping with reverse engineering, or a SOC analyst setting up custom detections*). Companies often launch internal projects with set timelines, but those deadlines can be suddenly accelerated due to shifting business priorities, client demands, or leadership decisions. This can lead to increased pressure on teams to deliver faster, sometimes with limited resources or preparation which absolutely NOBODY talks about and is HELL to go through when the timeline is TIGHTTTT. While the stress is hectic, it can be looked at as both a good thing for your experience belt, and a bad thing if stress as a result of high pressure environments is not managed properly.

That being said, working in unfamiliar environments can be simply solved by developing a training environment right? *Well, yes and no depending on the time you have XD.* This stems from similar concepts, but really takes a broad angle to it.

Remember how I said training is usually considered finished when the goal at hand whether it was exploitation or not is complete?

Of course here, its a little harder to build your entire cyber range for a stressful and maybe even new environment. So, Rather than manually crafting full applications to probe behavior, build focused testbeds that target the compiler and linker stages that matter.

In this case, spin up minimal Crystal programs (*and their C/LLVM IR equivalents*), vary optimization levels, toggle inlining and LTO, and compare emitted symbols and call patterns. By iterating on small, controlled samples you learn the concrete ways LLVM changes or translates high-level constructs into x64 code.

Sometimes, a well-built testbed is all it takes to give you just enough insight to push forward and not only in reverse engineering, but in any unfamiliar technical environment. It doesn't need to be perfect; it just needs to reveal the patterns.

Let's explore a common example where we actually do have to deal with compiler optimization, and find some room to exactly explore why the different perspective on the information you have in front of you matters.

Linker/Compiler Optimization

Everybody hates it but nobody can and will get rid of it. Compiler optimization at its core is the reason most code looks the way it does when you open it up in IDA. And yes, that is also why a single Go function call that has 1-2 lines of simple string calls looks like it has 300 lines of code inside of it….compiler optimization is both a beauty and a curse.

As reverse engineers, we understand that if we wanted to, we could look at the import table for when a symbol gets imported dynamically into the application and executed.

If the compiler and linker applied optimizations such as static linking or inlining, an imported function might be embedded directly into the binary. In such cases, the function may no longer appear in the symbol table, as its name is no longer needed. Instead, its instructions are placed directly into the **.text** section, occupying a unique address in the binary for execution.

- **Note**: I am primarily using the word ‘compiler’ to reference the language of choice here, but the one in charge of figuring out how code gets interlaced is primarily the linker. This component of a machine compiled programming language often has its own unique rules and definitions for being able to apply optimizations.

This renders such techniques (*like analyzing the import table and symbol table*) useless because the data does not reside there, and even then, the function name is often mangled, or even when unmangled, makes 0 sense to what the original function was unless it was something standard or is not app-specific.

So how do we handle this?

With our mentality of having understood some pattern recognition, we can take advantage of this and invoke a form of training I call Isolated Training.

Isolated Training

Isolated Training is a technique where a specific function or behavior is implemented in a minimal, standalone program to analyze how it appears in a binary under particular compiler and linker settings. This approach is especially useful when symbols are stripped or functions are inlined, making - as mentioned, traditional techniques…not work so well

Let's create a new example to demonstrate this: under static linking and compiler optimizations (*e.g., -O2, -static optimization modes used on GCC/Clang*), a call to **free()** may no longer be identifiable as it may be inlined, statically embedded, or renamed to a non-descriptive label (*e.g., sub_4012A0*), and its instructions placed directly in the **.text** section. As a result, no direct reference to free remains.

To get around this, we can write a small program that explicitly calls **free()**, compile it under similar conditions, and disassemble the resulting binary. From there, we locate the actual implementation of **free()** either in the disassembled code itself or by identifying where it's imported from (*e.g., the relevant libc or DLL*). This gives us a known reference for how **free()** appears in a binary under those specific settings. By comparing this reference binary (*ideally, one where* **free()** *is statically linked or inlined*) with the target binary, and using heuristics like instruction patterns and control flow, we can identify matching code-even when symbols are stripped or function names are obfuscated.

The goal here is not always going to be to find an exact match but to identify similar patterns. This is because deviations may not stem from compiler transformations alone; in adversarial scenarios, function-level similarities can reveal cases where malware authors or even single developers overwrite or hook known functions to bypass detection mechanisms like antivirus or EDR systems. This is why we say 'we can identify similar code'

Now, that being said, this method is not foolproof, and there is one flaw with it - it is definitely specific to where you get your reference piece. And this is the utmost important part.

We can analyze this by picturing the scenario for the reverse engineer using the free() reference to find similar code in a target binary. Imagine if they decided to pull a x32 version of free for an application written for x64. Or what if they had an ARM binary, but did not realize the function they were looking at was for a completely different version of ARM?

It is small niche details like this that can often make it time consuming to use such methods in a real world deployment. However, this is also why I decided to call it this form of '**training**' because we use it as a way to exercise how quickly we can recognize standardized functions, such as those in a system API, which should have universal and or similar structures across the same architectures/platforms or use it as a way to develop faster learning methadlogies. If an application was statically linking all of its functions, and for some reason, automatic heuristics weren't kicking in to the best of the ability, then we can analyze this ourselves.

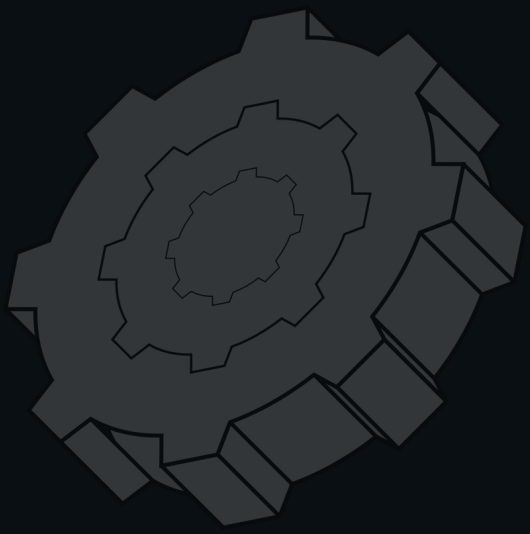
The technique is also helpful to teach yourself if you want to explore and compare the results of specific algorithmic implementations, or specific technical components to see how apps layout components internally. While it does primarily assist with the whole symbol table deal, it also can be used for a variety of other things. Especially for developmental purposes if you are somebody trying to figure out how to modify the function.

I know I mentioned we won't be going too practical, but to end this course off, I would like to throw a walkthrough of this and demonstrate exactly (*visually*) how this can be done, using our free() example from above.

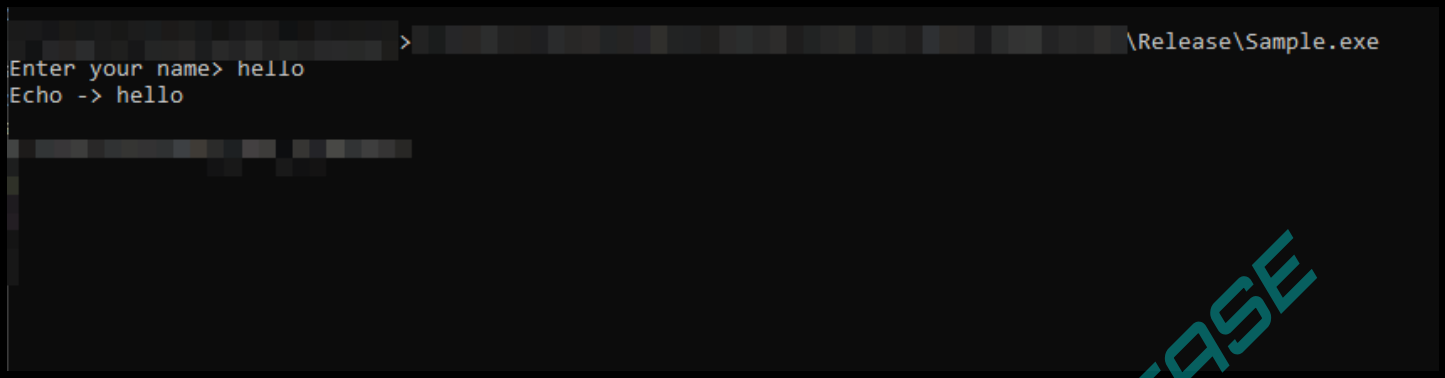
Isolated Training - Tracing free and using it as a code point of reference

As mentioned, we are going to take this application here, compile it using VC++ specifically with the code generation property on the project set to `Multi Threaded` from `Multi Threaded DLL` as we are looking for static linkage, we want to make sure the project is by technicality, multi threaded still, but does not use DLLs to rely on symbolic information. If you did not click on the link, the application is simple, in CLI format.

```
#include <iostream>

//// Haxxernote: it throws a security warning, its a bad unsafe function
int main() {
    char input[10];
    std::cout << "Enter your name> ";
    std::cin >> input;
    char* copy = (char*)malloc(strlen(input) * sizeof(char));
    strcpy(copy, input);
    std::cout << "Echo -> " << copy << std::endl;
    free(copy);
    return 0;
}

///// target: x64
///// type  : Release
///// pre_processing macros:
/*
        NDEBUG
        _CONSOLE
        _CRT_SECURE_NO_WARNINGS
        _UNICODE
        UNICODE
*/
```



When we compile this application and run it inside of a terminal, we get the following output upon giving it a set input.



Now, picture in this scenario, we are a reverse engineer who was tasked to rip open this application and find any vulnerabilities that may persist.

A note on binary auditing

If you are reading this lesson new to the world of reverse engineering or have not applied it to many areas in the real world, there are very particular aspects of reverse engineering, such as binary auditing, which are used to break out reports on the flaws that exist within binaries.

Binary auditing often involves analyzing well-known vulnerability classes such as buffer overflows by disassembling binaries to understand their typical patterns of exploitation. The same Isolated Training approach can be applied here: by writing small, controlled programs that allocate memory in different ways (*e.g., static vs. dynamic, heap vs. stack*), we can observe how these patterns manifest at the binary level. Comparing these reference binaries helps build intuition for identifying similar constructs in real-world applications, even when source code or symbols are unavailable. This is the portion of pattern recognition we have been trying to get across.

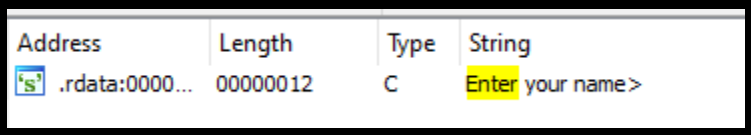
In the next section, we will take advantage of this to *try and look for a buffer overflow vulnerability* in the following application referenced here. The only thing is we will primarily be looking for it in a version of the application that was compiled to inline remote library functions.

Back to isolated Training

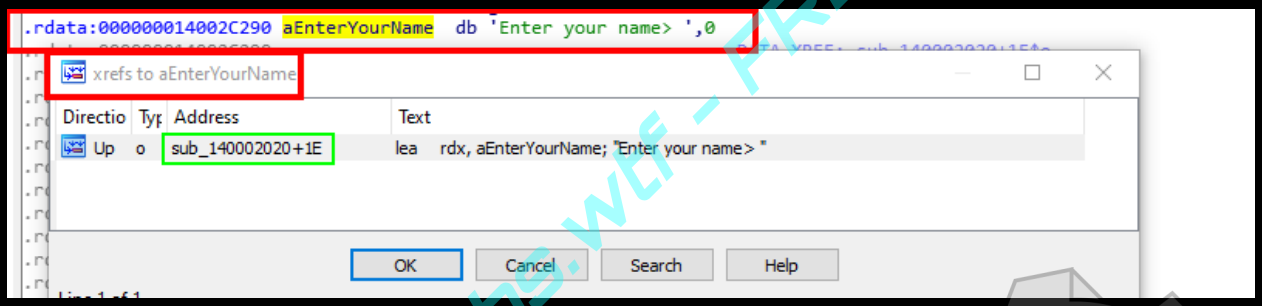
Now, let's take a look at this application, and see if we can't just use the string '*Enter your name*' to look for that buffer overflow we are expecting in this code once we disassemble the application in IDA-Pro *(version 7.0+)*. I say expect because if you look at the code, and have enough experience, the buffer overflow is obvious.

- **Note**: Expecting a BOF because if you are reading the code in the app, you'd notice the code commend for the pre-process VC++ macro '_CRT_SECURE_NO_WARNINGS' which disables CRT (*C Runtime*) security rules in the compiler.

When we do that, we see this string result pop up in the filtered strings window.



Then, we can double-click on the reference to which we get this location of the data which is noted when listing cross references (*highlighting over the data and clicking 'x' in IDA-pro*).



When we analyze sub_140002020 we find ourselves stuck in the Control Flow Graph (*CFG*) of IDA which slaps us in this code.

Since we developed this application or rather compiled it ourselves, we are aware that there is a free function somewhere in here. This can be used to identify the source of a buffer overflow, as we can track back the source buffer from where it was originally freed to where it was allocated! Of course we won't get this deep, and we are only here to look at the simplified patterns that signify the usage of a free() call being used.

That being said, to get a much higher level view of things, we can go ahead and run the pseudodumping tool within IDA to make sure we break down the full function logic.

```
1  __int64 sub_140002020()
2  {
3    __int64 v0; // rdx
4    __int64 v1; // rcx
5    unsigned __int64 v2; // rcx
6    _BYTE *v3; // rbx
7    char *v4; // rdx
8    char v5; // r8
9    __int64 v6; // rax
10   __int64 v7; // rdi
11   __int64 v8; // rax
12   __int64 v9; // rdx
13   __int64 v10; // rsi
14   void (__fastcall ***v11)(_QWORD, signed __int64); // rax
15   char v13[8]; // [rsp+20h] [rbp-28h]
16   __int64 v14; // [rsp+28h] [rbp-20h]
17
18   sub_1400023C0(&qword_140032370, "Enter your name> ");
19   sub_140002E40(v1, v0, (__int64)v13);
20   v2 = -1i64;
21   do
22     ++v2;
23   while ( v13[v2] );
24   v3 = sub_14000BD20(v2);
25   v4 = v13;
26   do
27   {
28     v5 = *v4;
29     v4[v3 - v13] = *v4;
30     ++v4;
31   }
32   while ( v5 );
33   v6 = sub_1400023C0(&qword_140032370, "Echo -> ");
34   v7 = sub_1400023C0(v6, v3);
35   v14 = *(_QWORD *)(*(_QWORD *)(*(signed __int64 *)(*(_QWORD *)v7 + 4i64) + v7 + 64) + 8i64);
36   (*(void (**)(void))(*(_QWORD *)v14 + 8i64))();
37   v8 = sub_140002280((__int64)v13);
38   LOBYTE(v9) = 10;
39   v10 = (*(unsigned __int8 (__fastcall **)(__int64, __int64))(*(_QWORD *)v8 + 64i64))(v8, v9);
40   if ( v14 )
41   {
42     v11 = (void (__fastcall ***)(_QWORD, signed __int64))(*(__int64 (**)(void))(*(_QWORD *)v14 + 16i64))();
43     if ( v11 )
44       (**v11)(v11, 1i64);
45   }
46   sub_140002B40(v7, (unsigned __int8)v10);
47   sub_1400029E0(v7);
48   sub_14000B888(v3);
49   return 0i64;
50 }
```

Data goes into v13

character range

Most likely pushing to memory

v3 = variable of interest, it seems like this is being cleaned up from the stack

Most likely freeing memory

This function is actually not that bad in its size compared to sometimes what you can see in the wild. Given that this is our main function though, we see a very simple

structure that resembles a lot of what our code actually is at its core. Let's break it down by color.

- Blue Highlighted text - All of these symbols were seemingly specific to the user controlled inputs that we defined in the top level of our main loop in the C++ program shown here. We could suspect this based on the way it was treated when being referenced throughout the function's scope.

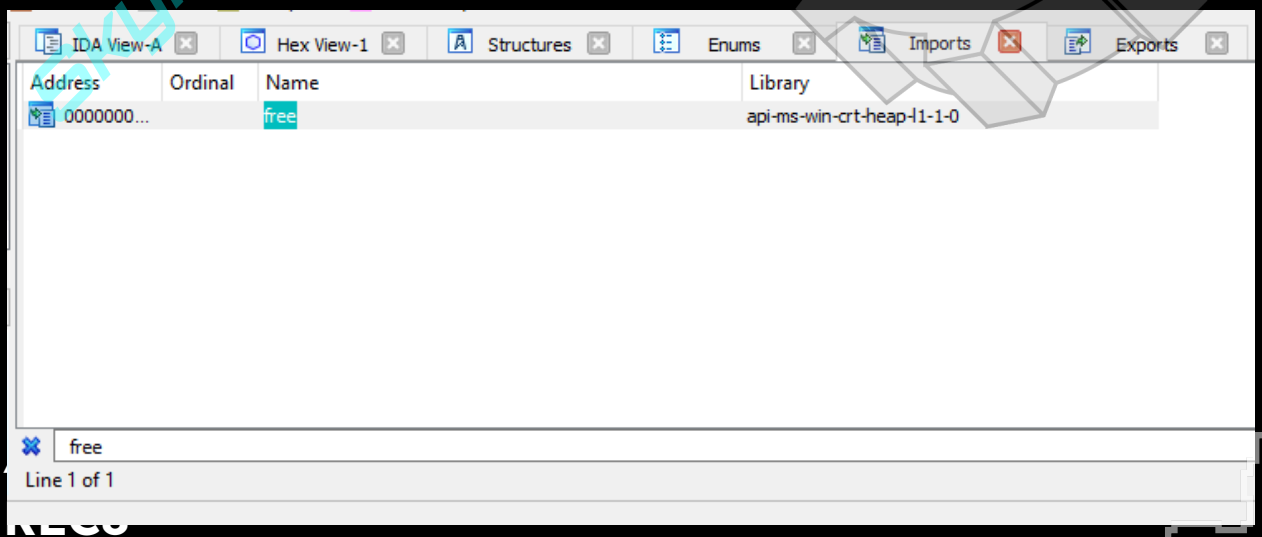
- Purple highlight - This is all of the variables that were important to track because they were relevant to the data being copied and referenced again after input. Additionally we see a variable referenced as v13 in purple which was used at the end of the function to a single function call (*any prospective hints?*)

- Green Highlight - Most likely memory operations happening that we currently do not have too much room to investigate with.

But we do not see the one thing we expect to find here - the free call.

Given that the version of the binary we are looking at was compiled with a project configuration that forces the compiler to call static inlining, everything is most likely to be statically linked, which means the need for function names or symbol names does not need to exist within the production binaries symbol table, and the need to reference symbols for import are not necessary. Alas: we have no function names now, even for standard C calls.

One way we can maneuver around this mess is to simply just compile the application in such a way that results in the binary using dynamic linking for the C runtime. Doing so results in seeing the `'free'` symbol pop up in the import table of the application when viewing it in IDA once again. We can use this as leverage for our own understanding.



As you can see, that function comes from a standardized library that is local to our system known as `api-ms-win-crt-heap-l1-1-0.dll` which you can also throw into IDA. We want to look at this file, because this file is where the function in our code above is being imported from, this way, we can get a much better and in-depth understanding of the structure within the free call.

- Note: Windows hates you working with System32 based files. That is exactly where this file is located on Windows 10. So if you are following along, make sure IDA is executed in an administrator mode, you are properly sandboxed or virtualized is the rather safer option, and go ahead and pop open the DLL inside of the directory.

When you do, go to the export tab and try to search for the keyword 'free' that we expect to be referenced within this DLL. We want to go to the exports because this is exporting function symbols which are read by our main application.

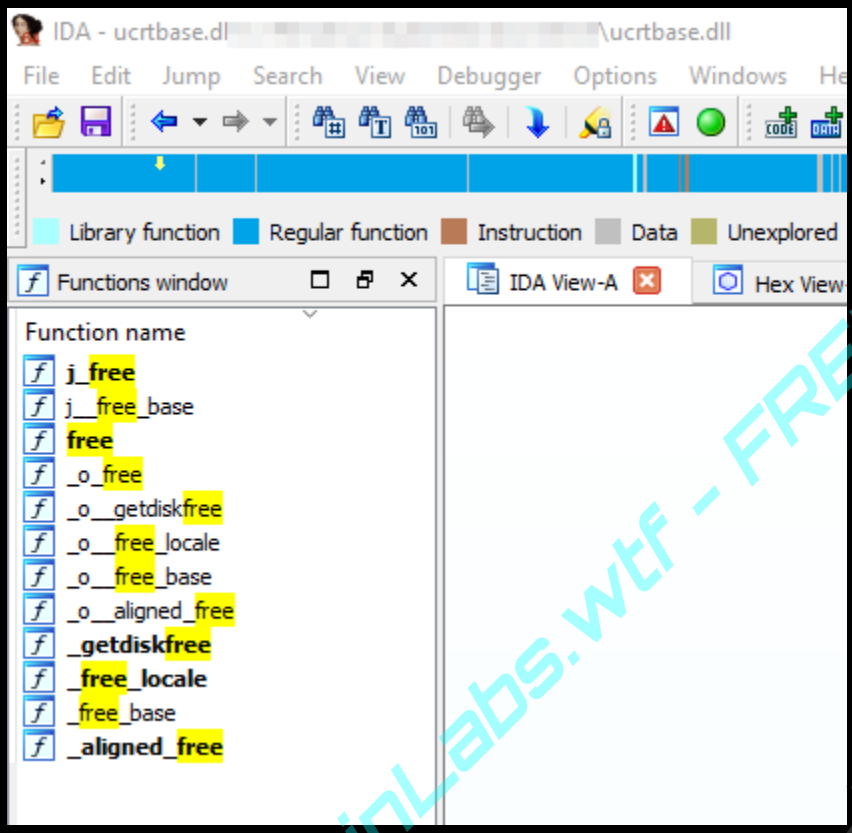
We see this:

```
.rdata:0000000180001603 aFree           db 'free',0             ; DATA XREF: .rdata:off_1800011C4↑o
.rdata:0000000180001608 ; Exported entry  25. free
.rdata:0000000180001608                 public free
.rdata:0000000180001608 ; void __cdecl free(void *Memory)
.rdata:0000000180001608 free            db 'ucrtbase.free',0    ; DATA XREF: .rdata:off_180001158↑o
.rdata:0000000180001616 aMalloc         db 'malloc',0           ; DATA XREF: .rdata:off_1800011C4↑o
.rdata:000000018000161D ; Exported entry  26. malloc
.rdata:000000018000161D                 public malloc
.rdata:000000018000161D ; void *__cdecl malloc(size_t Size)
.rdata:000000018000161D malloc          db 'ucrtbase.malloc',0  ; DATA XREF: .rdata:off_180001158↑o
.rdata:000000018000162D aRealloc        db 'realloc',0          ; DATA XREF: .rdata:off_1800011C4↑o
.rdata:0000000180001635 ; Exported entry  27. realloc
.rdata:0000000180001635                 public realloc
.rdata:0000000180001635 ; void *__cdecl realloc(void *Memory, size_t NewSize)
.rdata:0000000180001635 realloc         db 'ucrtbase.realloc',0 ; DATA XREF: .rdata:off_180001158↑o
.rdata:0000000180001646                 align 1000h
.rdata:0000000180001646 _rdata          ends
.rdata:0000000180001646
.rdata:0000000180001646
.rdata:0000000180001646                 end
```


Looking at this screenshot, we understand that the function being called is clearly noted by ida to be void **__cdecl free(void *Memory)** which points to an import

'ucrtbase' or Unversial C Runtime Base. So as you can imagine, we have to - once again: hop DLLs to find the meaning of free.

When we followed the SAME exact process in the previous DLLs (search for the popular keyword 'free'), we ended up getting this table of results.



Looking at these results, we notice a clusterfuck of functions we have to sort through!

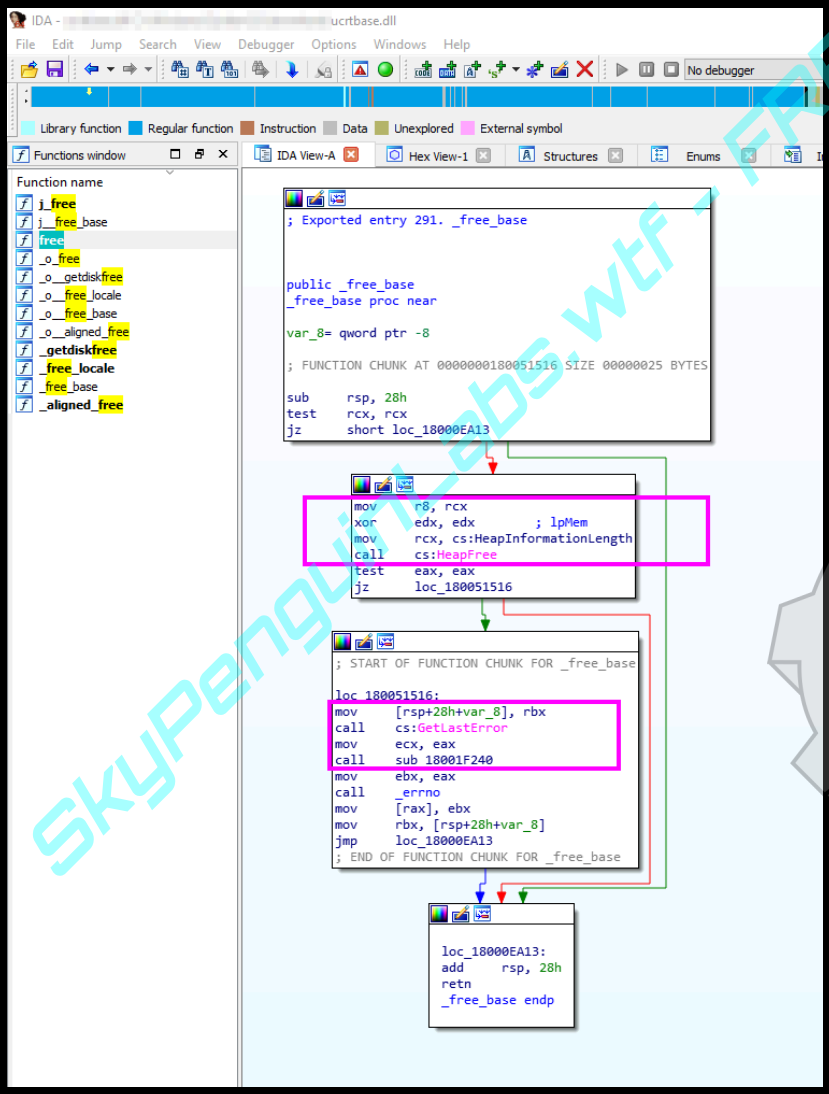
How do we know which one is the real call?

Well, we don't. We have to guess and start analyzing to figure that out. To **start** analyzing though, we can take the fact that our application calls the function *free* and we clearly see that in the box on the left in the functions window in IDA. Let's check it out in the image below.

```
; Exported entry 2212. free


; void __cdecl free(void *Memory)
public free
free proc near

arg_8= dword ptr  10h

mov     [rsp+arg_8], 0
mov     eax, [rsp+arg_8]
jmp     _free_base
free endp
```

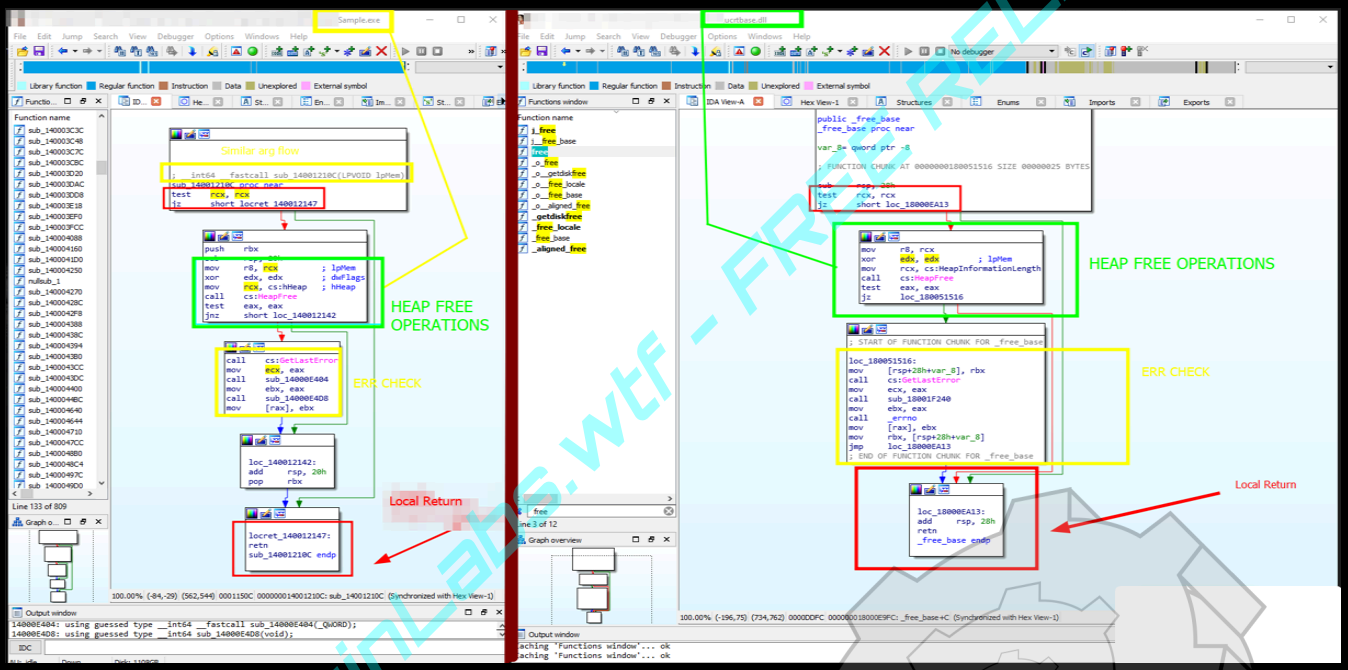
Ah yes, typical. A Thunk. Windows likes doing this a lot with functions it makes their own. Especially if they want to use their own system API!

Since this is a thunk setup to simply invoke the logic of _free_base, we can checkout the _free_base by double clicking on it and analyzing the CFG we get placed inside of below.



So it seems like what this call is actually doing, is implementing its own calling structure which relies on the Windows API. Which actually makes the naming scheme '`_free_base`' make sense here, since this call is using HeapFree once it receives the appropriate arguments.

If we open up our other application in IDA again which has its source code here, and go back to the main routine where we saw 'sub_140002020' at the end of its lifetime reach out to 'sub_14000B888' which calls 'sub_14001210C'. Viewing 'sub_14001210C' produces a similar output as the screenshot above. Let's compare them side by side.



(*P.S: you may need to zoom in to see minor differences*)

Putting them side by side, we got exactly what we wanted. We can clearly see that despite static linking, the SAME function is still technically used.

Lets conclude based on this training methodology.

Concluding Isolated Training - Collecting unorganized thoughts

Putting the two binaries side by side, we can see that there are virtually no significant changes after configuring the program to be statically linked. Apart from some minor differences, which are mostly attributable to typical compilation variations.

However, on the left side, we successfully identified the application's equivalent of the free() function. We can conclude this with reasonable confidence based on our

training methodology we discussed which was put into action when we compiled two versions of the same application, one using /MT (statically linked runtime) and the other using /MD (dynamically linked runtime).

In the statically linked (/MT) version, we noticed that the application no longer invoked free() via dynamic linking (expected). As a result, symbols like free() do not appear in the import table, making them harder to locate through standard symbol resolution. To work around this, we analyzed the dynamically linked (/MD) version of the application.

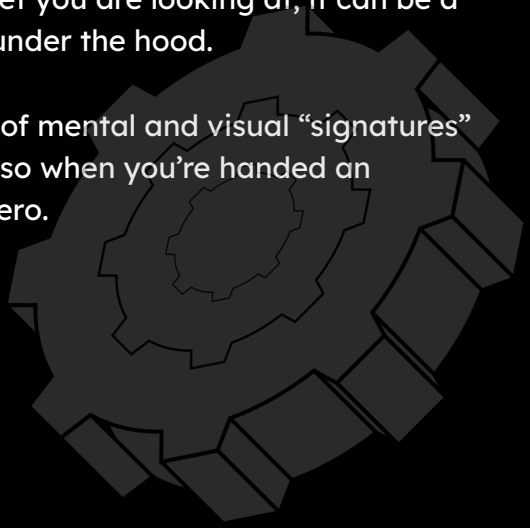
We located the call to free(), then traced it to the DLL from which it was imported. This revealed that free() is dynamically linked from ucrtbase.dll.

Opening ucrtbase.dll in IDA provided us with a standardized internal structure of the function. Once reverse-engineered, this can be used as a reference to identify equivalent inlined behavior in statically linked binaries. In other words, even though the static binary doesn't explicitly call free(), we can recognize similar instruction patterns and memory management logic by comparing it to the known structure of the dynamic version.

While we demonstrated that this type of methodology is applicable to dynamic linking scenarios, the genuine idea of being able to find something which uses a similar technology which is differentiable to the existing target you are looking at, it can be a huge help in understanding what exactly is going on under the hood.

The whole point is simple: You're building a set of mental and visual “signatures” by studying how known code translates into binaries, so when you’re handed an unknown binary in the wild, you’re not starting from zero.

Now let's conclude this article shall we?



Conclusion and Final Notes

SECTION: Section 0xFF | *Conclusion & Final Notes*

This article was technically long for something that was free, but as you can imagine, based on the writing alone, there is a reason as to why it was free. As reverse engineers, it is extremely important that we go through our brain and from time to time make sure we have a good collection of memory to go off of when trying to identify or work in new environments. But really that is just the only problem.

Many people swap over to different areas or even architectures when tackling RE roadpaths and feel like its just that specific architecture they don't get, and oftentimes, that is just due to how much lack of broad spectrum experience you have with it. Because to a specific point, when you get enough hands-on operational experience, your brain just clicks all of this into place. That's also why the idea of "*just follow this roadmap to get good at XYZ*" doesn't fully resonate with me. Sure, cybersecurity has some valid entry points, and fields like AppSec and even RE offer common starting paths. But when it comes to actually becoming skilled, there's no single defined route. You get good by being thrown into situations where you're forced to learn, adapt, and figure things out as you go.

Even these courses won't make you 'good' at cybersecurity, and the reason we admit it is because no certification or course out there is designed or can make you 'good' at something, unless it incorporates large amounts of core, real world non-simulated variables which add to the experience. Even then, it won't be a roadmap, it will just be a very specific and in-depth breakdown of how you can start off in the XYZ field. This is also due to how expansive it is.

So at a certain point, you have to ask yourself -when do I start brain hacking? Seriously. When do you?

Brains are important, mental health is even more important, and this lesson should have taught you how important pattern recognition is as a brain function- and how important it is to keep this in check due to the current shifts in the development spectrums around. The best way to do this is to keep even more up to date by finding or creating new ways to exercise the skill tree!

Finally, I hope you enjoyed the free 30+ page lesson and the new format! We are pretty damn proud of the progress we are making- it may be extremely small to everyone, but for us, it's small steps before big ones and you reading this is supporting exactly that! Thank you so much for taking a read at this lesson and I hope it gives you some insight into the content structures, or what type of content we produce at SkyPenguinLabs!