

# Codepoint-based Offset Portable Encryption



**Garret Stand**  
**Written on 9 December**  
**Cybersecurity Final**



TABLE OF CONTENTS

*A foreword ..... 1*

*How it works at the core ..... 1*

*The obligated parts of the writeup..... 2*

*THE FUN PART – algorithm inception & cracking the obfuscation ..... 2*

    Stage 1: Introducing COPE<sup>2</sup>..... 3

    Stage 2: Decrypting the encryption ..... 4

    Stage 3: Obfuscating Python like JavaScript... somehow ..... 5

*But, why???..... 6*

*Where to improve/go from here..... 7*

*Appendix..... 7*

# COPE

## A FOREWORD

This program/suite was written by me in the span of about a week through boredom, as well as this manual here, and as such, this is a brainchild of sorts of mine, so excuse my thoughts everywhere! Hopefully it makes this experience—if you want to call it that—better, and funnier; that’s the spirit of this anyways. Oh, and also, I apologize for the obfuscation and everything you are about to read about and/or have dealt with/are dealing with. Maybe this will be the next pizza program of legend... (sorry again Bart if you ever read this, at least you got it easy compared to this)

## HOW IT WORKS AT THE CORE

COPE is a multifaceted encryption algorithm with optional key protection. Short for “Codepoint-based Offset Portable Encryption”, or rather by the dictionary definition:

*cope*<sup>1</sup> /kōp/  
*verb - (of a person) deal effectively with something difficult.*

Once you unwrap the onion, through all the layers of obfuscation and encryption, you are met with a semi complicated algorithm underneath.

The following constructs are to be provided:

- An indexed string array with all permissible characters for the algorithm (A-Z, a-z, 0-9, !, ?, ., em space)
- An input string
- A key string (optional)

Given the input “this is a hidden message” and a key of “and this is the password?”:

- 1) Is there a key provided? If so, follow these steps:
  - a. For each character of the key, use the `ord()` function to find the UTF-8 ordinal number of said character.
  - b. Handling as an integer, add this value to an IV key variable.
  - c. Repeat until the entire key string has elapsed and each character’s value has been added to the IV key variable, returning 2362 for our example.
  - d. Find the modulus of the IV key by 66 and store it as the final key value, 52 for our example.<sup>i</sup>
  - e. Shift/rotate the character array by this value. This changes A being index 1 to A being index 14.
- 2) Do some safety checks to make sure the input is in the bounds of the permissible characters. In this case, it would pass. Otherwise, it’ll throw a `ValueError()`.
- 3) For each character of the input string, cycle thru the indices of the character array. Once there is a match, note down this number in the number array.
  - a. It is appended with an added offset of 16 (0x10 in hex) to ensure all resulting numbers will return an even number of hex digits, amongst some other logical enhancements.<sup>ii</sup>
  - b. This will return [75, 63, 64, 74, 29, 64, 74, 56, 29, 63, 64, 59, 59, 60, 69, 29, 68, 60, 74, 74, 56, 62, 60] for our example.

# COPE

- 4) The program will now take each number and convert it to hex, and then append each hex double to a hex string. keep this idea in the back of your head...
  - a. This will return '4b3f404a1d404a1d381d3f403b3b3c451d443c4a4a383e3c' for our example.
- 5) The program will now slice this blob of hex characters into groups of four into another array, before it can't anymore, and it'll just settle with two.
  - a. This is where 3a comes in, for reasons explained shortly.
  - b. This is also where the 0x head of the hex digit, trimmed in step 4, will return attached to the four/two-byte chunk.
  - c. This will return ['0x4b3f', '0x404a', '0x1d40', '0x4a1d', '0x381d', '0x3f40', '0x3b3b', '0x3c45', '0x1d44', '0x3c4a', '0x4a38', '0x3e3c'] for our example.
- 6) The program will then take these codepoints, literally evaluate them to an integer<sup>iii</sup>, that then gets fed into chr(), which spits out the Unicode character at the passed codepoint.
  - a. This will spit out the very strange output of, erm, '龕曙ᵀ霽嶺甌肱歎ᵉ歎霨烽'ⁱᵛ
  - b. As per footnote 2 and 5a, it is important that an even number of hex digits are present, as two/four are safe and even three is, but one-byte remainders are not, most single digit codepoints are weird control characters in Unicode that as noted, my code does not like. Found that out the hard way.
- 7) Anyways, this will then get spat into the base64- wait, no, base85 encoding method<sup>v</sup>, and print its output to stdout.
  - a. This returns our final encrypted string of  
'<gCBsfr{a^faIv1<Di}6y@2DayW\_Ei;kAV0v5MrVxZ}9I'.

To decrypt the string, its essentially the other way around. The decryption algorithm converts the base85 to the raw Unicode gibberish, converts it to hex, squishes it back into a single hex string, take out the doubles, convert them to octal, do the appropriate shifting/offset operations and pull each character from the character array. The next challenge, however, is figuring out how the code works...

## THE OBLIGATED PARTS OF THE WRITEUP

As per the rubric, I have to include the difference between what “currently exists” and my own twist with it. I feel my program here is quite similar to, and I actually got some inspiration from, cryptii.com. It is less of a specific algorithm, more of a site that you can combine many encodings and encryptions with, of which inspired me to do the same thing with the various encodings present here, but I just put my own divinely inspired hellish twist to it, both with how fundamentally janky but functional it is, and pairing with the journey you are about to dive into now...

## THE FUN PART – ALGORITHM INCEPTION & CRACKING THE OBFUSCATION

In spirit of the suite's name and of the past pizza program, I felt that just making this encryption algorithm wouldn't be fun enough for the reader, so I decided to put 10x more effort than was taken writing the original program into obfuscating, (recursively) encrypting, and encoding the program just for the fun factor. Depending on how much of a glutton for punishment/spirit of fun you are/have as the reader, let it be known that the entire programs key—if you will—is contained here for un-obfuscating it, so consider it

# COPE

as a spoiler warning, I guess. Given the amount of effort I put into the obfuscation, and totally not to satisfy my evil side of watching the CTAE teachers suffer again (totally not), I would advise—in the “spirit of fun”—to take a shot at trying to crack it prior to reading this. If you manage to at least get somewhat far, you will earn my respect and the right to read this guide without guilt. But if you couldn’t be bothered, the key, or rather lockpick, to the kingdom is fully present herein. All stages of the decrypted/decoded/deobfuscated code, along with some tools, the toolchain to build your own obfuscated programs, and libraries, are present in `src/`.

## STAGE 1: INTRODUCING COPE<sup>2</sup>

 *(the source file `release/encrypt.py` and `release/decrypt.py` applies here)*

That being said, we’ll start from the first layer of the onion then progress to the core, the final running code. I am sure you are asking how in the Lord Mother’s name that I ended up making this what’s already a monstrosity of a program suite run off of an extremely long hexadecimal string in the source files, seemingly just-in-time interpreted by an already unclear one-liner. Well, I now introduce to you, one of many inventions/PEP8 violations present here, COPE<sup>2</sup>, or the COPE Code Obfuscation Portable Encoding. Presenting itself as the first obstacle to your true understanding of this enigma, I ended up making a way to encode the stage-2 (yes, there’s stages, keep reading) wrapper and a one-line decoder and just-in-time interpreter. It actually found its initial use in the roots of the raw program code and evolved, but I’ll get into that later.

How it works is actually surprisingly simple when explained. The only precondition for a given code file to use the encoding algorithm with its default values (offset 128/page size 2, these will be explained shortly) is only containing two-byte Unicode characters below codepoint 0x7F, but even if it contains four-byte characters/characters above the codepoint ceiling (for some reason), the encoding usually is adjusted to accommodate relatively easily<sup>vi</sup>. The encoding is simple, its actually contained in a one-liner itself.

Given the input `‘print(“Hello World!”)’` as a string:

- 1) For each character in the string, use `ord()` to find the ordinal number of the said character, and add 128 (or whatever the user selects/algorithm enforces) as the *offset*.
  - a. The offset is to prevent just converting the hex to text, that’d be too easy.
- 2) Convert it to hex and do the similar stylistic 0x head trimming in step 4 above.
  - a. Do the appropriate padding operations to ensure a consistent hex digit length. This is referred to as the *page size*. It is 2 by default, but if you have characters whose ordinals exceed 2 hex digits, the algorithm will enforce a minimum of however long of a hex digit is required by the highest character. Per current Unicode-16 standings, this means a maximum of 5 is possible.
  - b. This returns `‘f0f2e9eef4a8a2c8e5ececefa0d7eff2ece4a1a2a9’` for our final encoded string.

This can all be contained into the following one-liner, given `toencode` is the input variable containing a string conforming to default values, and `encoded` is the output destination variable:

```
>>> for i in toencode: encoded += hex(ord(i)+128)[2:].zfill(2)
```

# COPE

You can actually test this out with `src/tools/cope2encode.py` and `src/tools/c2c.py`, to either encode a string yourself or compile your own script to use COPE<sup>2</sup>! The above isn't very far from the implementation in the libraries that power COPE<sup>2</sup> either; check `src/lib/cope2.py:30`.

The decoding, seen present in the second line of each of the final code files here, works like this; given the input of the output given above and an initialized buffer variable as a string (z1 and z2 in the source respectively), and the aforementioned default values<sup>vii</sup>:

- 1) Using some magic for/range statements (the first time I've ever realistically used these outside of APCSA, these are essentially for/i loops...), take each two-byte hex double in the input string, and convert it from hex to a decimal integer<sup>viii</sup>, and subtract it from 128 to remove the offset.
- 2) Use `chr()` to convert it to its character counterpart, then append it to the buffer.
- 3) Once the buffer string equals to the length of the input divided by two, meaning its fully converted, execute the buffer string with `eval()`<sup>ix</sup>.
  - a. The reason it's done like this inside the for loop as opposed to outside of it is solely for the fact that I can keep it on one line like this. Do I even have to explain why I did it this way still?

In the one-liner (well, really two, since I can't condense the input/buffer variable declarations on the same line as the decoding/JIT evaluation because you can only skin a python so much...) fashion of it all, the encoded program above with the wrapper logic dictated here would come out as:

```
>>> z1, z2 = 'f0f2e9eef4a8a2c8e5ececefa0d7eff2ece4a1a2a9', ''
>>> for i in range(0, len(z1), 2): z2 += chr(int(z1[i:i+2], 16)-128);
... exec(z2) if len(z2) == len(z1)/2 else None
```

Easy enough, right? Now if we just switch the `exec()` call to a `print()` call in the files, voila, you get the decoded code in your console! (or you can just be lazy and pass the file as input to `src/tools/cope2decode.py` with the decompile flag. :P)

## STAGE 2: DECRYPTING THE ENCRYPTION

*(the source file `src/encrypt/stages/stage_2.py` applies here)*

After applying the logic above to the source file for the encryption engine<sup>x</sup>, you are left with what looks like a hydra of further COPE<sup>2</sup> encoded code and... COPE<sup>not squared</sup> encrypted code? What the hell is `h[:]=h[w:r(h)]+h[u:w]???` Well, addressing that horrid, obfuscated logic routine is getting a bit ahead of ourselves, but let's pay attention to what we know so far; COPE and COPE<sup>2</sup>. Looking at the `stage_2` file here, `z3` seems to be what's clearly COPE encrypted... something. Well, let's try and paste it into the decryption program! (Given that you escape all the reserved characters before feeding to the console, see footnote V)

```
> python3 release/decrypt.py ";h>Y@kgeg2w&9SL;f%K7jI-gKvEh)l;heGIoUq}Xvf+@...
IDEdDmEGDmDjHhElHhHgHiEmHhHiHiEeHiEgHdEmHhEeHhEdHiEfHhEhHeEiHeEgEkHdHhEiHe...
cope echo "Ummm... that's not right?" at 09:08:56 PM
```

Well, that's not right. Doesn't look like code nor hex. Well, judging off what we know, there could be a chance that this data could be encrypted with a key. But where could

# COPE

that key be? Well, let's look at what else we have here in terms of crackable codes. We have z1 and z4... z1 is typically reserved for the JIT interpreter to use for code so let's check z4, it also doesn't seem to have the `exec()` present in its interpretation loop either. Well, let's toss it into the decoder!

```
> python3 src/tools/c2decode.py "f9eff5a7f2e5a0e7e5f4f4e9eee7a0e3eceff3e5f2"  
🔒 COPE² Decoder  
Success! Decoded string:
```

you're getting closer

```
🍏 📁 cope echo "Ahhh, there we are!"
```

```
✓ at 09:09:49 PM ⓘ
```

There's our key! Let's toss the key into the decryption program again and take another shot, shall we?

```
> python3 release/decrypt.py -k "you're getting closer" ";h>Y@kgeg2w&9SL;f...  
fab1a0bda0a7e5b9e5e4e6b0e5e6e6b2e6b4e1b0e5b2e5b1e6b3e5b5e2b6e2b4b8e1e5b6e6...  
🍏 📁 cope echo "Agh, more COPE²?"
```

```
✓ at 09:08:56 PM ⓘ
```

Well, congratulations, you just found another layer of COPE² encoding. I promise you're getting there though... Before we recurse any more though, I want to stop and take a minute to look at the rest of the code we seemed to forget. Well, spoiler alert—if it wasn't already obvious, that's the decryption code. Specifically, it loads the obfuscation dictionary for the decryption (you'll learn more about this in a second), the decryption key, decrypts the executable code for the encryption program, which encoded in COPE², runs thru another one-liner decoder, of which finally executes it. How many bloody times could I have said decrypt in that sentence?

## STAGE 3: OBFUSCATING PYTHON LIKE JAVASCRIPT... SOMEHOW

📁 *(the source file `src/encrypt/stages/stage_3.1.py`, `src/decrypt/stages/stage_3.1.py`, `src/encrypt/stages/stage_3.2.py`, `src/decrypt/stages/stage_3.2.py` apply here)*

With where we are now, the code is starting to look at least somewhat like Python code, though only somewhat... remember what I said earlier about `h[:]=h[w:r(h)]+h[u:w]`? We're plagued with many similar sights in both files... though notice that aside from a few differences to automate it, the decryption file is pretty much the same code featured in stage two from before. Oh, and remember what I said about an obfuscation dictionary? Well, let's go ahead and crack open z1 and see what we get...

```
> python3 src/tools/c2decode.py "f9eff5a7f2e5a0e7e5f4f4e9eee7a0e3eceff3e5f2"  
🔒 COPE² Decoder  
Success! Decoded string:
```

```
import base64  
from ast import literal_eval as b  
import argparse  
a = argparse.ArgumentParser(description="cope encryption suite - encrypt")  
...
```

```
🍏 📁 cope echo "This is getting somewhere!"
```

```
✓ at 09:12:14 PM ⓘ
```






# COPE

Well, there's our obfuscation dictionary! This is the code that basically makes all this unreadable code readable to the interpreter, executed as basically a JIT translation dictionary to load all of the lambdas and et cetera into memory. We can basically do the same, substitute lines 1-3<sup>xi</sup>,—the decoding and execution blocks—of each file, and now we really have a decently readable Python file!

As said earlier in stage 1, this early raw program code saw the creation and first implementation of COPE<sup>2</sup> encoding in the codebase, of which I created to hide the obfuscation dictionary if anyone happened to get this far unwrapping the code, which to be fair, if you got this far it would be of no avail to me, but the sentiment is there.

Anywho, the obfuscation dictionary really makes things look a lot more readable—at least with the help of the VS Code hover definition hints. If we rewind a little further, I actually got inspired to do this off of a meme I saw of someone making Python code in C using compiler preprocessor statements, #DEFINEing everything they needed to make the compiler understand it. This is sort of what's happening here. There's no define statements here, but you may see there are a copious amount of lambda statements, basically just a really compact function comparable to an => arrow function in JavaScript, that in a way just remap all of the readable methods to random letters. On top of that, all of the hardcoded values and other initial things such as the argparse initialization and imports are all contained herein too. Now, if you just transplant everything using this knowledge, and shuffle some things around to combat the other, more humanly, obfuscation methods you're left with...

 **Congratulations!** Your final source files! (also present in  `src/decrypt/main.py` and  `src/encrypt/main.py`) Your journey, of either reading or following along, is complete! Now you can see how it all works on the inside as per the “how it works” section!

## BUT, WHY???

Well, I truly got my inspiration to not only make the insanity that is COPE but the onion around it due to my close friend/basically brother Tyler Peters, whom of which is also in ICS with me, partaking in his creation of an equally janky encryption algorithm with matrices he was learning about in Pre-Calculus at the time. I saw what he was doing, spoke out loud the idea of using the numbers his program would spit out as Unicode codepoints, then I suddenly said, “never mind, I'm claiming that”, and after about 30 seconds of thought, I had the entire idea of this planned out in my head, the evil plan if you will. I approach this with the same mentality that I did the pizza program approximately one year ago. I did what I was told here, but it was boring, so why not add a massive twist to it? I did with adding a full CUI and database driver to my pizza program and I did it here creating what is, to my awareness, a brand-new way to (albeit jankily) encrypt/encode/obfuscate Python code, and a toolchain to do such! I am not a massive fan of obfuscation outside of stuff like this as I am a proponent of open source (this is all going on GitHub some-time soon at the time of writing), but this is mostly if not entirely reversible, especially with the right knowledge, so I see it as okay. At least I was nice enough to not compile the Python to now require a decompiler here as well? I have at least some heart, and limit to the amount of energy I'll put into this, I think...



# COPE

## WHERE TO IMPROVE/GO FROM HERE

Well, as seen blaringly throughout this entire document and codebase, especially in the copious number of footnotes present in the appendix below, that there are plenty of bases where the program can be improved on. For the sake of the rubric, I'll just list them here.

- 1) See footnote I, key bypass issue  
Can probably be alleviated by a check after the modulus is calculated (1d) and possibly providing a negative rotation?
- 2) See footnote V, console compatibility  
Figure out way to make console-friendly output for sake of convenience
- 3) Improve consistency of obfuscation, just redo it all once I have a stable toolchain created
- 4) Put the effort into beautifying the CUI, if you will
- 5) Create a cross client secure messaging type of thing? Super far-fetched but possible. The premise of the encoding here makes it really easy to crack but it's worth a try.
- 6) Make an obfuscation engine, like stage three, with something like interpreter hooks to dynamically change things on the fly? I don't even know how I'd dynamically interpret the code without executing it much so as to make the changes without breaking things.
- 7) Probably a ton of other things I'm missing that I'll come across on and fix at some point.

As it stands right now, there exists a COPE and COPE<sup>2</sup> library with inline documented APIs that one can use for implementing the algorithms in their program, with compilation programs/routines available as well so if one desires, one can build their program into. I am working on a toolchain so a “build system” of sorts can be available so one can build their own modular CI workflow for obfuscating and/or encrypting their program with as much flexibility as they like, and even other modules too to plug in... basically just make for Python that uses the COPE suite! However, time constraints forbid this happening by the due date, so it'll just come as its own package at some point.

## APPENDIX

---

<sup>i</sup> This does present a theoretical abnormality where if one were to enter a key whose summated octal values added up to an integer perfectly divisible by 66—or an uppercase B, which possesses the Unicode codepoint of 0x42 or decimal 66—one would be left with a key integer of 0, which is default and will not protect the processed string. However, I do not think I am protecting government secrets with this algorithm, nor will anyone use straight up “B” as a password (at least wisely) so... I think I'll be fine.

<sup>ii</sup> This also avoids the issue of a null character being possible along with some Unicode control characters that my code did *not* like, as about to be explained. How the hell one character ended up causing a memory leak and crashing VS Code is beyond me but that is beyond the scope of this write-up.

<sup>iii</sup> Evaluation of a user inputted string is probably unsafe, and it goes without saying not to not do this. This is the worst way I could have chosen to convert a string to an integer, but I guess it adds to the effect of this entire program's existence. I feel as if we're due to learn this sooner or later in Cyber anyways...

<sup>iv</sup> Probability states that since a good majority of the Unicode codepoint space is dedicated to the Chinese-Korean-Japanese block and their sprawling amount of ideographs, and given such, you'll get some random Asian characters a lot of the time, especially with the four byte codepoints, partially why I

# COPE

chose four bytes as opposed to two, to increase confusion if one were to decode the base85 output string (explained later on). Given the linguistic properties of these Asian languages, this does form a valid and somewhat comprehensible sentence and most random characters will too. That being said, if my program curses you out in Chinese, I am not responsible.

<sup>v</sup> A very funny Intellisense misclick crawling the [base64](#) module led to my accidental discovery of [base85](#). My next question is why though... It has also been occurred a few errors with base85 output using reserved shell operators/characters but as long as you escape everything you should be fine. If you're still reading this you know how to do that, I promise you.

<sup>vi</sup> This is easy enough as just taking the `zfill()` present in the encoding one-liner, changing the padding to go up to 4 (or however many) hex digits, and changing the decoding code to read each 4-digit hex chunk as opposed to just reading doubles, particularly in the `range()` contained in the for loop (`range(0, len(z1), 2)` to `range(0, len(z1), 4)`), as well as accommodating the very janky progress checking code to consider the 4:1 length ratios between input and output as opposed to 2:1 (`len(z1)/2` to `len(z1)/4`). This is essentially what's done in the toolchain and library when it does its automatic sanity checks to make sure you don't over/underflow anything with your offset/codepage/input combination. I cannot escape math no matter how hard I try, even down to my least favorite subject matter in my most favorite. Brilliant!

<sup>vii</sup> To get a rough idea of how this works outside of the default values, just replace every mention of 2 with your page size and 128 with your offset in question. The library safety checks the values and accommodate if need be.

<sup>viii</sup> The way I did this with `int(hex, 16)` (where hex is your to-be-converted input) practically eliminates the issue presented in step 6 of the main algorithm/footnote III, and I could have implemented it that way, but in the spirit of the jankiness of the code, I chose not to in the main algorithm, but did it this way here so I could get away with not importing the `literal_eval()` routine from `ast` anyways, further mystifying the one-liner at first glance anyways. There's a million ways to do anything in Python anyways!

<sup>ix</sup> The entire point of the last footnote then continues to get defeated here as I use `eval` again. What goes around comes around...

<sup>x</sup> The decryption engine doesn't apply to any further sections after stage #, "Obfuscating Python like JavaScript... somehow".

<sup>xi</sup> This was supposed to be 1-2 but—and you may have noticed this far in—I didn't do my two-var one-line trick I did with the rest of the COPE<sup>2</sup> implementations... the reason why is that you are looking at the first COPE<sup>2</sup> implementation that's ever touched the overall COPE codebase! This was so early on I noticed after the fact but in the process of wrapping the onion by hand (before the tools present here were made, this was all by hand in the Python console, with some praying to the Python gods involved) I couldn't be bothered to change it.

END  
THANK YOU FOR READING  
WRITTEN WITH LOVE,  
GARRET STAND :)

**CRITICALLY ACCLAIMED:**

**"Ms. Whitlock just going  
to have to hard cope"**

**- Tyler Peters**

**"I think I just had a stroke  
trying to understand this"**

**- Nicolas Ash**

**"We all love Python"**

**"On god"**

**- Leo Hao and  
Garret Stand**

**COPE**

