# Flare-On 10, Challenge 10, kupo

## Challenge Description

Did you know the PDP-11 and the FOURTH programming language were both released 53 years ago, in 1970? We like to keep our challenges fresh here at Flare-On, in-line with the latest malware trends.

## Files

| Filename | Size | SHA256 |
|---|---|---|
| mog.dsk | 121,452,544 bytes | 089799e05b69763a12c14a31774ba15e47ae1e95904b59e467909f3e3d446ec3 |
| forth.tap | 6,594 bytes | 9b8efbbd3d1a2738c8c7a859f488f5736734857fb969286a0e58d2ef9e9dd8c9 |
| README.md | 1,867 bytes | 0f253fb761e172c359784beda75302aacba0b6d9b7f8a607e83a7197dac9fc18 |
| mog.cfg | 599 bytes | 7ee37622b285808ac1d67c01b6be093068eb95f880042d34efce4e1ccda41116 |

The file `README.md` contains some challenge setup descriptions. `mog.dsk` and `mog.cfg` are emulator image and configuration files. `forth.tap` is a tape image file.

## High Level Summary

- The challenges files comprise a 2.11BSD operating system disk image and a tape image
- The tape image contains two files, a second step README and a compressed forth environment executable
- The forth environment contains standard and custom words
  - Words can either be a sequence of more words (defined in forth) or be native assembly code
- The relevant challenges words are **secret**, **decrypt** and **decode**
  - **secret** pushes a pointer to the crypted flag string specification (length and string pointer) on the data stack
  - **decrypt** applies a multi-byte XOR cipher on the string specification on the data stack with the key being read from behind the word (using parse)
  - **decode** applies a prefix-sum modulo 256 on the string specification on the data stack
- A forth word solution that also prints the flag: `secret count decrypt p/q2-q4! 2dup decode type`

## Environment Hints / Setup

The challenge file **README.md** gives an introduction about the challenge environment. It tells us to pick one of our physical PDP-11s or the SIMH emulator, for which the configuration file mog.cfg is provided. We should read it in order to figure out how to boot. Great, RTFConfig!

Once we are able to boot, we are supposed to hit enter on the bootloader and then press `ctrl + d` at the root prompt. Login creds are `root / Flare-On`.

Of course we need to figure out how to mount the tape drive TS0 and extract the challenge file. The next stage help file may then be reached via `cat /dev/rmt12`.

### PDP-11 Emulator

We can get a free copy of the SIMH emulator from GitHub and consult the SIMH manual. SIMH can also be installed via `pip install simh` on Linux systems.

## Stage 1: Booting & Challenge Files Acquisition

Taking a look at the supplied SIMH configuration file `mog.cfg`, there are comments that need us to add things. With some reading, we can figure out that it's not enough to just enable the TS11 tape controller and drive, we also need to attach the tape.

This can be achieved with `attach ts0 forth.tap`. In order to directly boot the system up, we can also add `boot rq0` at the end.

See the edited mog.cfg file. We can boot the newly configured system with `pdp11 mog.cfg`.

### Booting

```
44Boot from ra(0,0,0) at 0172150
:       <- hit return here once
: ra(0,0,0)unix
Boot: bootdev=02400 bootcsr=0172150

2.11 BSD UNIX #115: Sat Apr 22 19:07:25 PDT 2000
    sms1@curly.2bsd.com:/usr/src/sys/GENERIC

ra0: Ver 3 mod 6
ra0: RD54  size=311200

phys mem  = 2097152
avail mem = 1873216
user mem  = 307200

hk ? csr 177440 vector 210 skipped:  No CSR.
ht ? csr 172440 vector 224 skipped:  No CSR.
ra 0 csr 172150 vector 154 vectorset attached
rl 0 csr 174400 vector 160 attached
tm ? csr 172520 vector 224 does not exist.
tms 0 csr 174500 vector 260 vectorset attached
ts 0 csr 172520 vector 224 attached
xp ? csr 176700 vector 254 skipped:  No CSR.
erase, kill ^U, intr ^C
#      <--- hit ctrl+d here
Fast boot ... skipping disk checks
checking quotas: done.
Assuming non-networking system ...
checking for core dump...
preserving editor files
clearing /tmp
standard daemons: update cron accounting.
starting lpd
starting local daemons: sendmail.
```

```
Fri Aug  4 22:40:31 PDT 2023


2.11 BSD UNIX (curly.2bsd.com) (console)

login: root
Password:
erase, kill ^U, intr ^C
#
```

## Acquiring the next README

Now it's time to get the challenge files from the tape.

```
cd /tmp
cat /dev/rmt12 | tee desc.txt

Welcome to MoogleForth!

This may be an interesting challenge for you.  The next file on this tape is an
executable Forth environment which contains a secret and the means for decoding
and decrypting that secret.  You'll need Ken Thompson's password, which I trust
you'll be able to find.  Beyond that, you'll need to figure out how the various
Forth words want their input, which will require some detective work on your
part.

You can solve this challenge with nothing but the tools available to you on a
standard 2.11BSD system, such as nm and adb (and if you're not familiar, you
should read their man pages).  You may find it easier to use a more modern
disassembler, though you'll need to be able to extract the file from the tape
for that.  And, of course, you need to know (or learn) how Forth works.  On the
bright side, I did leave all the symbols in the executable for you, I'm not a
monster.

The Forth environment is fairly stripped down, but where possible, I've tried
to conform to standard behaviors for all the standard words.  You can find much
documentation for Forth 2012 online, with an excellent reference at:

http://lars.nocrew.org/forth2012/alpha.html

Some caveats:

- There's very little error checking here.  You'll probably crash a lot if you
  provide unexpected input (or fail to provide expected input).  Sometimes,
  especially if you underflow the stack, the interpreter can get confused.
  You can always ctrl-C to quit if 'bye' isn't working for you.

- I've basically implemented just enough of the words to build this challenge.
  In particular, there's no compiler system, so you can't write your own colon
  definitions.  Sorry!  There was only so much time.
```

```
- Some of the words are defined in assembly, most are defined as more Forth.
  This may make disassembly interesting, but doable.

- It is dark.  You are likely to be eaten by a grue.
```

So the next file on the tape is our long awaited Forth environment that we need to grab the secret from. We also need to know Unix super guru Ken Thompson's password. This is `p/q2-q4!`, a nice to remember chess move notation that was pretty uncrackable back in those times! For the sake of completeness, here is the full passwd line found with googling. `ken:ZghOT0eRm4U9s:52:10:& Thompson:/usr/staff/ken:`

We definitely want to solve the challenge just with onboard 2.11BSD tools. Why deny ourselves that historical sensation by leveraging modern tools? No way.

We are going to be eaten by a grue. Not just once.

### Getting the Forth Environment running

The next file is a compressed a.out executable.

```
# cp /dev/rmt12 c10.Z
# uncompress c10
# chmod +x c10

# file *
c10:    executable not stripped
desc.txt:       English text
```

In case something goes wrong, we can *rewind* the tape with `mt -f /dev/rmt12 rewind`.

## Stage 2: Experimenting with the Environment

Off to our first start.

```
# ./c10
MoogleForth starting. Stack: 3802
```

Awesome. Hard to describe the amount of despair having absolutely no idea at all about everything here and so forth(!).

That is the moment we can:

1. cry
2. apply for sales jobs
3. both
4. grab a shovel, because we're going to dig deep into unknown territory

Never ever having been exposed to a forth environment before, this was a *painful* series of baby steps. One at a time...

Since I was totally clueless, I went back to the 2.11BSD shell and took a look at `nm` and `adb`.

```
# which nm
/bin/nm
# which adb
/bin/adb
```

Reading the manpages is always helpful and a must for `adb`. Train your brain to think `[4m` and `[m` are invisible, helps a lot if you can manage that.

Dumping the symbol names and offsets with nm gives a first idea about what we have at our hands here. Output is shortened for readability, see nm.txt for the full output.

Having no idea about what is normal and what is relevant, we have to just pick something and go with that.

```
# nm c10
000644 t _bugoff
005250 d _secret
001126 t _xor
001114 t _xor_xt
000226 t decode
000212 t decode_xt
000324 t decrypt
000306 t decrypt_xt
003376 t execute
003360 t execute_xt
004174 t hash
004164 t hash_xt
000510 t key
000476 t key_xt
000000 t main
002766 t parse
002752 t parse_xt
005162 t secret
005146 t secret_xt
```

Now some of these have interesting names: Secret, everything with crypto, hash, key, execution. You name it, you disassemble it!

nm's man page states, that the `t` in the second column means that symbol is in the text section, `d` in the data section. There's only `_secret` in the data section, it seems.

While we're at it on the shell, we can also dump the challenge binary to extract it from the emulator environment with `od -h c10`. It is important to realize that PDP-11 is a 16 bit little endian word machine. So all hex words from `od` are displayed in a swapped order. To *repair* that with CyberChef:

1. From Hexdump
2. Swap Endianness, Data format raw and word length to 2 bytes.
3. Save the challenge binary

It can now be analyzed with other disassemblers like IDA Pro.

## Stage 3: Forth Baby Steps

The challenge binary provides a Forth REPL-like interpreter. All input is taken as a `word`, which is like a function definition. Each word can be implemented as assembly code or as a sequence of other forth words.

Some ground truths can be read up on here. Lovely page for citations like `"Without a traditional operating system, Forth eliminates redundancy and needless run-time error checking."` Yeah, who needs run-time error checking anyways if you're good with stack housekeeping... haha. But that's a joke one may only understand by crashing the environment roughly 2**16 times in a row.

Oh. And don't forget, there are two stacks.

1. The data (call frame stack). Register `r5` points to the top of that one
2. The return stack R. Register `sp` points to that stack's top.

The challenge author also provided this link to look up on standard forth words. That page though isn't easy to digest as a forth newcomer. 😉

To get a list of all words available in this challenge, use the word `words`

```
words

secret s" >number place cmove move find words pad tib here hold #s # #> <# holdend
holdptr ." . base cr space bl #tib >in 2rdrop rdrop 2r@ r@ 2r> r> 2>r >r execute
1/string /string skip scan (parse) parse-name parse post-parse pre-parse word
source do-word quit abort ; immed compare max min 0<> 0> 0< invert > < <> = 0=
until if goto rot tuck nip 2over over 2swap swap 2drop drop 2dup dup zero c! c@ !
@ xor or and * /mod cell- 1- - cell+ 1+ + accept key emit decrypt decode type
count align sp@ bye
 ok
```

Now that we know which words are available and some resources to look up definitions (even if we have no idea what we're doing yet), we can start to crawl around. Smashing the stack has never been easier.

But first things first: forth uses the reverse polish notation also known as postfix notation. And each and every input/output parameters is going through a stack. Let's try something as simple as adding two numbers and getting the result back. Sounds easy? Sure, if you know how to do it...

Taking a look at the glossary, we find that `+` is a forth word. It's also available in our environment, as can be seen from the word list. We do encounter something called a `stack notation` for word definitions, which basically describes what and how many parameters a forth word expects on the stack as input and what it will leave on the stack as a result.

For `+` it's `+ ( n1 n2 -- sum )`

- n1 and n2 are expected to be on the data stack
- `+` does its magic and leaves a sum on the data stack

While addition is commutative, even in forth, the order of parameters pushing is important. Also stack housekeeping.

In order to do a simple addition of `3 + 4` and get the result, we do `3 4 + .`. What does that mean?

- Push 3 on the data stack
- Push 4 on the data stack
- Execute the word `+`
- Pop the result from the data stack with the `.` word. Everything is a file^H^H^H^H word!

```
3 4 + .
7  ok
```

Let's try something else: `/mod` (divmod). According to the documentation, its stack notation is `/mod (n1 n2 -- n3 n4)`. It calculates a modulo division, leaving result n3 = quotient and n4 = remainder on the stack once it's done.

```
3 5 /mod . .
0 3  ok
(0 rest 3)
7 5 /mod . .
1 2  ok
(1 rest 2)
```

Awesome. Don't be afraid to fail - A LOT - in between those insights. I have left out countless of hours of miserably failing to call anything without core dumps all over the place. For sake of readability and logical structure, I leave those in my `night of pain` notes.

Defining and working with strings was something to get used to. The linked documentation's author clearly didn't use MoogleForth, it seems. Strings - or more accurate string specifications - consist of a length value and an address to the characters of the string. They are defined by the word `s"` (not `."` as mentioned in the link).

```
s" hi" . .
2 3805  ok
```

But how can we make forth print a string? With the word `type`!

```
s" hi " type
hi  ok
```

Awesome. Everytime you meet ok, clap yourself on the shoulder. Unless you messed the stack up. But that's okay too!

Here is a list of standard forth words with their stack notation blatantly copied & pasted together.

```
calc stuff
- + - * as expected ( n1 | u1 n2 | u2 -- n3 | u3 )  Add/substract/multiply   n2 |
u2 to n1 | u1, giving the sum/difference/product n3 | u3.
- 1- 1+ ( n1 | u1 -- n2 | u2 ) Subtract/add one (1) from n1 | u1 giving the
difference/sum n2 | u2.

stack stuff
- rot ( x1 x2 x3 -- x2 x3 x1 ) Rotate the top three stack entries.
- tuck ( x1 x2 -- x2 x1 x2 ) Copy the first (top) stack item below the second
stack item.
- nip ( x1 x2 -- x2 )Drop the first item below the top of stack.
- 2over ( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 ) Copy cell pair x1 x2 to the top of
the stack.
- over ( x1 x2 -- x1 x2 x1 ) Place a copy of x1 on top of the stack
- 2drop ( x1 x2 -- ) Drop cell pair x1 x2 from the stack.
- drop ( x -- ) Remove x from the stack.
- 2dup ( x1 x2 -- x1 x2 x1 x2 ) Duplicate cell pair x1 x2.
- dup ( x -- x x ) Duplicate x.

char and string stuff / something with c-addrs
- c! ( char c-addr -- ) Store char at c-addr. When character size is smaller than
cell size, only the number of low-order bits corresponding to character size are
transferred.
- c@ ( c-addr -- char ) Fetch the character stored at c-addr. When the cell size
is greater than character size, the unused high-order bits are all zeroes.
- emit ( x -- ) If x is a graphic character in the implementation-defined
character set, display x. The effect of EMIT for all other values of x is
implementation-defined.
- type ( c-addr u -- ) If u is greater than zero, display the character string
specified by c-addr and u.
- count ( c-addr1 -- c-addr2 u ) Return the character string specification for the
counted string stored at c-addr1. c-addr2 is the address of the first character
after c-addr1. u is the contents of the character at c-addr1, which is the length
in characters of the string at c-addr2.

a-addr / cell stuff
- ! ( x a-addr -- ) Store x at a-addr.
- @ ( a-addr -- x ) x is the value stored at a-addr.
- cell+ ( a-addr1 -- a-addr2 ) Add the size in address units of a cell to a-addr1,
giving a-addr2.
```

## Stage 4: What's the challenge about anyways?

Having so much fun learning the basics of forth let's you totally forget about why we're here in MoogleForth in the first place. Something about a flag? Let's try to find hints about it...

Now that we have a first grip on some easy standard words, can we spot some of the custom/challenge specific ones? Together with the output of `nm` that we saved in a write-only part of brain memory, we recall/infer that the words `decode`, `decrypt` and `secret` sound promising. But what are they? How do we use them? What parameters do they expect? That's a yes to all of those questions. NO CLUE?! (yet)

## Figuring out (custom) word definitions

In other, more feature-rich forth environments, there may be words like `view`, `see` or `'` (tick) that may give hints about the definition of a word. But ... this is a reverse engineering challenge after all.

So what do we do? Right, play around and crash everything.

```
0 decode
 ok
```

WOW! It said ok! But wait, what did it decode? Why don't we see anything?

```
42 decode
Memory fault - core dumped
```

Yikes. It's broken now, can I give it back? These are just a few examples of failed attempts, resetting the environment. Sometimes rebooting the simulator, rewinding virtual tape drives. What I'm trying to say here is: You crash, you get up, you run against that wall again. Eventually it will break in. Or you find and take a door. Whatever comes first.

After umpteen restarts of the challenge, it looks like the word `decode` takes a string specification. But it's selfish and doesn't give anything back.

It took some time to digest the stack machine workings but after some time...

```
s" hi " 2dup decode type
hQq ok
```

What does that mean?

- We put the string specification of `hi` on the stack with the word `s"`
- `2dup` duplicates the specification (length and string address)
- `decode` does whatever it does with it, selfishly not giving anything back
- Haha, but since we saved a copy of the string specification on the stack, `type` can use that to print out the decoded string

Great, progress. But what's `hQq` ffs? Is it base 2-85? Hmm.

```
s" 4eKk " 2dup decode type
4dOo ok
```

```
s" 1337 " 2dup decode type
1dNn ok
```

Still no idea. Looks like the first character stays the same. Then things go uphill. Will it be nice to us if we feed it Ken Thompson's password?

```
s" p/q2-q4!" 2dup decode type
pBo`5 ok
```

No. Clever people may instantely realize here what `decode` does. I didn't though (yet).

What about `secret`?

```
secret
 ok
```

Yeah, I get it. It's a secret.

```
secret .
2728  ok
```

Yeah, thanks for the fish...

Enough guesswork, time to start wielding `adb`, the shipped 2.11BSD debugger.

## Stage 5: Debugging Baby Steps

Take an hour to read the man page of `adb`, if you can. Or be a better googler to find resources like this before you solve the challenge the hard way. 😉 Whatever suits your style.

Start debugging the challenge binary with `adb c10` (or whatever you named it).

Most important concept to understand here is that pretty much all you do follows this general scheme:

`address [, count] command] [;]`

Write it on a sticker. We need that. I also parsed 65.3% of the man page into my write-only excerpt here.

We can do all funny things with `adb`. Including disassembling words, and debugging of course. It takes some time to get a hang around it, but practice... is a process.

Let's try to inspect some words like secret, decode and decrypt.

### Inspecting Secret

Let's disassemble `secret`, that sounds important.

```
adb> secret
               05162
adb> secret?
secret:        jsr     pc,_const
```

How does that work?

- `secret` is used as a (symbolic) address (remember the scheme above?). And thanks, Dave, for leaving symbols in it.
- Using it just on it's own prints the value of secret. Numbers with leading zeros are octal (base 8) values.
- `?` is a means to dereferencing the value of secret as an address.
- So `secret` is a custom challenge word, that seems to be implemented in assembly.
- `jsr` is a `jump to subroutine` branch opcode to `_const`. What's there?

```
adb> _const?i
_const:        mov     *(sp)+,-(r5)
adb>
_const+02:     mov     (r4)+,pc
```

- Something is pop'd from the sp stack (return address stack) and dereferenced (`*`)
- And that dereferenced value is pushed onto the data/r5 stack
- The second line is basically a `return` like opcode. Register `r4` often carries the value to be put into the program counter register `pc`

If you're not comfortable with the assembly and cpu registers, this webpage is a great resource to look things up. I literally spent 80% of the challenge on that page and the adb manpage. 😉

Back again to the output above. What was on top of the sp stack? To understand that, we have to trace back to the `jsr` opcode that lead us here. Here's the description of it: `JSR Jump to subroutine: -(SP) ← Reg; Reg ← PC; PC ← Src`

So the `jsr` opcode located at the address of the word `secret` pushes the address following that opcode on the sp stack, register `pc` stays the same and the address of the symbol `_const` is put into the program counter (i.e. executing the branch).

But what was the value located just behind the `jsr` opcode at secret?

```
adb> secret+04?o
secret+04:     05250
adb> secret+04?d
secret+04:     2728
```

Wait. We've seen that number before when executing the word `secret` in forth and then pop'ing a value from the data stack!

What's the meaning of it though? With the power of `adb`, let's find out:

```
adb> 2728?o
lastword+042:   056
adb> 2728?x
lastword+042:   #2e
```

Uh-huh. The value 0x2e. Great. From `nm` output of symbol names, we remember to have seen one lonely data section symbol named `_secret`. Are they siblings?

```
adb> _secret?o
lastword+042:   056
adb> _secret=
                05250
```

Okay. Looks like the word `secret` puts the address of `_secret` on the stack. Let's dump all there is

```
adb> _secret,20?X
lastword+042:   #2ed51b         #c3787c2f       #dac22e75       #32787bd6
                #23d8d97d       #318a863d       #2ccc2d81       #c47c74d6
                #273ff682       #345760d8       #e9c7d032       #7b18f21
text address not found
```

If we count things together, it adds up that there are 0x2e (decimal 46) bytes behind the value 0x2e, which may indicate that this is a length value. This could be our secret, likely the flag bytes in an encrypted/encoded form.

Fiddling around some more, reading documentations and pondering about a sales career, we stumble upon the word `count`. Given an address to a string on the stack, count returns a string specification of that string.

Back in the forth environment:

```
secret .
2728  ok
secret count . .
46 2730  ok
```

So:

- secret pushes a pointer to something we imagine to be the encrypted flag bytes (string)
- we pass that on to count

- count pushes the string specification on the stack (length and string/c-address)

Hey we can `type` it now!?

```
secret count type
UxC/|BZu.x2V{X#}Y
1=L,-|DVt?'vW4X`Gi2P1! ok
```

Beautiful garbage. But progress!

Back to the other words: decode and decrypt

## Stage 6: Understanding Decode

By now, we already dream in forth. Let's go crazy with it:

```
s" 11223344" 2dup decode drop dup c@ . 1+ dup c@ . 1+ dup c@ . 1+ dup c@ . 1+ dup
c@ . 1+ dup c@ . 1+ dup c@ . 1+ dup c@ .
49 98 148 198 249 44 96 148  ok
```

What are doing here?

- Create a string specification for the string `11223344`
- Duplicate the specification with 2dup
- Let one clone be eaten by decode
- drop the length from data stack. Who needs that anyways
- Duplicate the address of the - now decoded - string
- Execute word `c@` to have it take one character value from the address on the stack. It consumes one clone of it by doing so, but nicely enough puts the character value on the data stack.
- Pop that character value from the data stack to print it out (49 = "1")
- Use the word `1+` to increase the dup'd string address
- And go nuts with repetition, because - Dave didn't implement a way to compile words.

If you are less tired than I was when doing that, you'd might realize already what decode does. I still didn't. 😉

Because `adb` didn't get enough love and attention from us yet, let's disassemble decode instead of guessing.

```
adb> decode?i
decode:         jsr     r4,_docol
decode+04:      bne     0177754
decode+06:      ble     decode+030
decode+010:     bne     bye_xt+04
decode+012:     beq     0177732
decode+014:     swab    r0
```

Is this assembly?!? At least the very first line looks legit... Some `jsr` branching to `_docol`. Grue? Is that you?

So why not disassemble `_docol`.

```
adb> _docol?i
_docol:         mov     (r4)+,pc
```

Say what? It's jumping to the dereferenced value of r4. What was in r4? The `jsr` in `decode` put the (address) value of `decode+04` into r4 before branching. So basically speaking - all that `_docol` does is jumping to the address in r4 and increasing its (addres) values.

But isn't that crap following there? It is, if we keep staring at the disassembly. We have to realize that this is not assembly what we're looking at, but other forth word (addresses to be precise)! That may be quickly googled up by searching for `_docol` or a full night of pain. These word addresses are jumped to in a list with r4 storing the next address. A serialized execution of words, if you will.

So whenever we see the `jsr` to `_docol` at the disassembly beginning of a word, we *now* know that it's a word implemented in forth, not assembly. Let's see what that definition holds:

```
adb> decode+04?p
decode+04:      zero
decode+06:      tor
decode+010:     dup
decode+012:     if
decode+014:     decode+052
decode+016:     swap
decode+020:     dup
decode+022:     cat
decode+024:     rto
decode+026:     plus
decode+030:     twodup
decode+032:     swap
decode+034:     cbang
decode+036:     tor
decode+040:     oneplus
decode+042:     swap
decode+044:     oneminus
decode+046:     goto
decode+050:     decode+010
decode+052:     rdrop
decode+054:     twodrop
decode+056:     exit
```

Hey that looks halfway nice and readable! `?p` interpreted the address values as symbols. Thanks to Dave once more from me. Can't repeat that often enough. Imagine this without a symbol table...

Now we can disassemble each of these and also single step debug through all of them. Once we do, we realize that decode ain't much magic.

- It ingests a string specification from the data stack
- Loops over each character
- During each iteration, the byte value of the last character is added to the current one
- The byte sum is stored back at the position in the string (overwriting the value at this index)
- Since it starts with `zero` as the *last character value* the first byte is never changed. Wait, we already observed that when playing around with the word in forth!
- This looks like a prefix sum mod 256

We can implement that in python or forth easily ourselves!

```
def decode(input: bytes) -> bytes:
    result = bytearray(len(input))
    cur_byte = 0
    for i in range(len(input)):
        cur_byte = (input[i] + cur_byte) & 0xff
        result[i] = cur_byte
    return(bytes(result))
```

Because why not, let's for fun and giggles stuff the secret into decode

```
secret count 2dup decode type
?akY_+WXENuwmDxP0w`b;J ok
```

Hm. Does it ring a bell (pun intended)? No.

Okay, we now know where the secret is and what values are stored there. We also know what decode does. We know it doesn't help us with the secret (yet). What's left is decrypt.

## Stage 7: Decrypting Decrypt

We can and should definitely feed whatever comes into our mind into the word decrypt. Chances are, it segfaults all day long. Seriously does.

What would we expect for a decrypt function to take as input?

- At the very least, it may want a buffer and its length to do its magic on.
- For it to properly be classified as decryption - cough - it needs key material.
- That could be hardcoded or somehow provided as input as well.
- Usually with some length information about the key, too.

But how? What is the order of parameters? Let's try to understand it's assembly

```
adb> decrypt
                                        ; prologue
decrypt:        mov     r2,-(sp)        ; push r2 on sp stack
decrypt+02:     mov     r3,-(sp)        ; push r3 on sp stack
decrypt+04:     mov     r4,-(sp)        ; push r4 on sp stack
```

```
decrypt+06:     mov     r5,-(sp)        ; push r5 on sp stack

decrypt+010:    jsr     r4,bl           ; push r4 on stack
                                        ; push address decrypt+016 on sp stack
                                        ; jump to bl. Disassembling bl yields,
                                        ; that it puts the constant value 32
                                        ; (040) on the data stack
decrypt+014:    decrypt+016
                                        ; r5, the data stack pointer, points to
                                        ; 0x20 / 32 / 040 after returning from bl
decrypt+016:    mov     (sp)+,r4        ; pop r4 from sp stack (restore r4 value
                                        ; saved @ +010)
decrypt+020:    jsr     r4,parse        ; push r4 on stack, push address
                                        ; decrypt+026 on sp stack and jump to parse.
                                        ; parse's stack notification says, it
                                        ; expects a char delimiter as input (bl)
decrypt+024:    decrypt+026
decrypt+026:    mov     (sp)+,r4        ; pop r4 from sp stack (stack housekeeping?)
decrypt+030:    mov     (r5)+,r3        ; pop r3 from data stack. This is the length
                                        ; of the key string parsed from REPL
decrypt+032:    mov     (r5)+,r1        ; pop r1 from data stack. This is the
                                        ; string/c-addr of the key string parsed
                                        ; from REPL
decrypt+034:    mov     (r5),r2         ; put top of data stack into r2 / does not
                                        ; remove it.
                                        ; that's the length part of the input
                                        ; string specification
decrypt+036:    mov     02(r5),r0       ; put the second to the top of data stack
                                        ; into r0 / does not remove it
                                        ; this is the string/c-addr part of the
                                        ; string specification
decrypt+042:    mov     r3,-(sp)        ; push r3 on the sp stack (length of key)

                                        ; outer XOR loop starts here
decrypt+044:    cmp     r2,r3           ; set flags based on r2 - r3
                                        ; r2 = remaining length of input string
                                        ; to decrypt
                                        ; r3 = len of key
decrypt+046:    bgt     decrypt+054     ; if r2 > r3, do a full key cycle
                                        ; i.e. we still have more input characters
                                        ; to process than the key is long

decrypt+050:    mov     r2,r3           ; if not, set r3 to r2, the remaining bytes
                                        ; from input string
decrypt+052:    beq     decrypt+076     ; if there is nothing more left, we're done
                                        ; here. Move along

decrypt+054:    sub     r3,r2           ; r2 = r2 - r3
                                        ; substract the key length (to be decrypted
                                        ; in the next inner loop cycle) from the
                                        ; remaining amount of bytes of the input
                                        ;string

                                        ; inner XOR loop
```

```
decrypt+056:    movb    (r0),r4       ; fetch the next character from input
                                      ; string c-addr
decrypt+060:    movb    (r1)+,r5      ; put the current key byte into r5 and
                                      ; increase key c-addr
decrypt+062:    xor     r4,r5         ; xor both, result in r5
decrypt+064:    movb    r5,(r0)+      ; mov byte result back into the current
                                      ; input string c-addr position
                                      ; and increase r0, ie. move r0 to the next
                                      ; input string byte
decrypt+066:    sob     r3,decrypt+056; substract one and branch, jump if
                                      ; r3-- > 0, inner XOR loop
                                      ; r3 = remaining key byte length

decrypt+070:    mov     (sp),r3       ; put top of sp stack into r3 (a new key
                                      ; cycle, if you will)
decrypt+072:    sub     r3,r1         ; r1 = r1 - r3
                                      ; r1 = remaining characters to decrypt
                                      ; from input string. r3 = key length
decrypt+074:    br      decrypt+044   ; jmp, restart outer loop

                                      ; epilogue, stack housekeeping
decrypt+076:    tst     (sp)+         ; pop key length from sp stack
decrypt+0100:   mov     (sp)+,r5      ; pop from sp stack r5
decrypt+0102:   mov     (sp)+,r4      ; pop from sp stack r4
decrypt+0104:   mov     (sp)+,r3      ; pop from sp stack r3
decrypt+0106:   mov     (sp)+,r2      ; pop from sp stack r2
decrypt+0110:   mov     (r4)+,pc      ; put (r4++) into pc, i.e. jump back
                                      ; to the caller
```

If you want to follow along with debugging, because static analysis can be tough, do make use of adb. Here's a list of useful breakpoints. `:b` behind the symbolic address means set a breakpoint, `$r` at the end means execute the command `$r` upon breakpoint hit. `$r` just displays all register values.

```
# adb c10
adb> decrypt:b $r
adb> decrypt+026:b $r
adb> decrypt+044:b $r
adb> decrypt+056:b $r
adb> decrypt+070:b $r
```

Then run the program with `:r`, landing in the MoogleForth. Apply MoogleForth words to get this running.

```
s" test" decrypt key type
ps      0170000
pc      0324    decrypt
sp      0177574
r5      07326
r4      02540
r3      06670
```

```
r2      06413
r1      07
r0      02544
decrypt:        mov     r2,-(sp)
breakpoint      decrypt:        mov     r2,-(sp)
```

A debug log up until the first xor can be found in decrypt_debug.txt. It seems as if single stepping with `:s` sometimes seems to execute two instructions. Ah well, our static analysis theory has been backed up, so we can start to sum things up now.

## Stage 8: What we know so far

- The word `bl` is executed at decrypt+010 yielding the constant value 32 (040, 0x20)
- This is put on the data stack as input for the word `parse`
  - Parse's stack notation `( char "ccc<char>" -- c-addr u )`
  - This means, the value 32 is a delimiter character code (it's the space character code!)
  - Side note: The description found @ nocrew is horrible to read for forth noobs
- So `parse` works on the forth REPL line, parsing the characters following the `decrypt` word up until the space character (parameter)
  - It will put back the string specification (string/c-addr and the length of the string up to the delimiter character) on the data stack
- Decrypt implements a multi-byte XOR cipher and uses the `parse` output as key characters and key length

## Grand Final: Putting it all together

We can retrieve the flag either with a Python script or with Forth words.

### The Python Way

With all we know, we can locate and decrypt the flag from the challenge file.

### The Forth Way

How do we connect the dots between the words `secret` (the crypted flag), `decrypt` and `decode`?

- `decode`
  - Takes input from data stack: a string specification (len + ptr)
  - Transcodes in memory
  - Does not *return* anything to the stack.
  - Applies an adder with feedback from previous character (prefix sum mod 256)
- `decrypt`
  - Works on input on the data stack: Works a (byte) string specification
    - To feed it with the `secret`, we leverage `count` to put a string specification of it on the data stack
  - Takes addition input from Forth REPL via `parse`: the key up to first space character and key length

    - *Returns* a string specification on the data stack
        - Yeah I know, technically it never removed it from the data stack
    - Applies a multi-byte XOR cipher

Writing this down after having crawled out of the challenge's rabbit hole, it obviously only makes sense to first `decrypt` and then `decode` just by looking at each one's expected input and output.

With all the Forth fairy dust ingested by now, we can solve it with its own tooling

```
# ./c10
MoogleForth starting. Stack: 3802
secret count decrypt p/q2-q4! 2dup decode type
ken_and_dennis_and_brian_and_doug@flare-on.com ok
```

Explanation

- `secret` is a pointer to the (byte) string specification of the crypted flag
- `count` converts that to a string specification.
- `decrypt` takes that from the stack (len and address) for its decryption buffer
- `decrypt` uses `parse` with `bl = 32 (space)` to get the decryption key and key length from the forth word line
    - The challenge readme hinted that we need Ken Thompson's password.
    - This is the place!
    - It would be a shame to have it had cracked and remain unused.
- `decrypt` yields a string specification back an stack
- `2dup` duplicates it, so that we can print it after decoding
- `decode` decodes the string in memory (no output, no return value), but *cosumes* one two word string specification from the stack
- `type` prints the string specification saved with `2dup`

## Flag

Flag: `ken_and_dennis_and_brian_and_doug@flare-on.com`

PS: Wow, what a ride that has been. Thanks, Dave, for this awesome challenge. It was great fun (and sometimes pain) to be exposed to this historical unix environment.

PPS: `126 42 xor emit 104 dup emit 7 - emit 110 dup dup emit 3 - emit 29 xor emit` Dave!