Flare-On 10 Challenge 10: kupo

By Dave Riley

Overview

This challenge is partly a response to the feedback from the challenge I wrote last year, which was apparently "too easy". Careful what you wish for! This challenge is fairly involved and requires some reading and research. It's part lore, and part OSINT, but mostly proper reverse engineering, if on a somewhat different system than most people are useful.
This challenge is also somewhat inspired by our move to Google, where Ken Thompson (the "father" of Unix, and co-creator of the Go language) works. The purists among you will note that the image is a BSD system, not a stock Bell Labs Unix system, which is a fair cop, but it was much easier in the end to go with 2.11 BSD, so I did.

Approach

There are a few ways to solve this challenge, including purely static methods that don't require running the supplied system image. It is, however, probably most straightforward to go through it the way I'm about to show.

The supplied files

The challenge comes with a premade disk image for a 2.11BSD system (in case you don't already have one, but any 2.11BSD system ought to work), a configuration file for SIMH's PDP-11 emulator to run said disk image, a tape image containing the challenge, and a README file (copied above).
Technically, you don't need the system at all to solve the challenge; the .tap format is a fairly common, straightforward and well-documented one, so you can find code out there to extract the contents of the tape or write your own fairly easily and proceed from there.
However, assuming you're not doing this the hard way, let's proceed with trying to boot the system and go from there.

Booting the system

SIMH is an emulator that's widely available for most operating systems under the sun, and is fairly light on resources (it may not be the most efficient emulation, but even on a Raspberry Pi it's still considerably faster than the original systems it emulates). You can install it with most package managers, usually with the emulators for all systems it supports, but occasionally piecemeal. You'll want the PDP-11 version for this.
It can be started on its own, and you could probably boot the supplied image even with its defaults, but you'd have to figure out how to attach all the devices, which can be a headache, so it's best to use the supplied configuration file which sets nearly everything up for you. Per the README, you can do this with pdp11 mog.cfg which will result in the following:

```
PDP-11 simulator V3.12-2
Disabling XQ
Enter "boot <disk>" to boot.  Make sure to read the config file and manual!
sim>
```

How cryptic! We don't know what the disk is! Maybe mog.cfg is worth examining. Let's take a look at that:

```
; Make it a PDP-11/44 with 2MB of RAM and an FPU (because 2.11BSD didn't have
; a working FPU emulator until 2006 (!)
set cpu 11/44 fpp 2m

; Disable some of the default devices we don't use.
set tm dis
set rp dis
set rx dis
set rk dis
set hk dis

; Enable the hard drive and attach it.
set rq en
set rq0 ra80
attach rq0 mog.dsk

; Enable the TS11 tape controller and drive.
set ts en

; Normally we'd have this here, but I'm being mean and making you use the SIMH
; console so you know how to attach the tape later.
echo Enter "boot <disk>" to boot.  Make sure to read the config file and manual
```

Okay, so we can see that mog.dsk is attached to rq0. That's enough to boot, though for the curious, there is a wealth of knowledge to be gleaned from the SIMH manual on the RQ device type. You'll need the manual for some of the next bit as well, but for now, let's go back to SIMH and enter boot rq0.

```
sim> boot rq0

44Boot from ra(0,0,0) at 0172150
:
```

Clearly, this is another prompt. In this case, it's the prompt expected from the 2.11BSD bootloader, which is the first stage of things. At this point, you could look up what to do with the bootloader, or you might just hit enter and see what happens:

```
: 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: RA80  size=237212

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
#
```

You're booted into single-user mode now (which you might not know without consulting the installation instructions), and while you could do the rest of this challenge in single-user mode, it won't be very fun, especially if you don't set the terminal types (which aren't initialized until multi-user mode).

To finish booting to multi-user mode, simply hit Ctrl-D. This is actually in the install instructions, though it would have been fairly common knowledge back in the day; Ctrl-D is EOF, which causes the single-user shell to exit and init to take over, which is still true in a lot of modern Unix and Linux distributions if you boot into single-user mode, which hasn't been the default for a long time.

```
# 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 22:44:15 PDT 2023


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

login:
```

You have reached a login prompt. Login as root using the password given in the README.

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

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

We're ready to begin.

Reading from the tape

So if you explore around a little, you'll find some things that are similar to modern Unix/Linux systems, though there is a lot that isn't (including a lot that just isn't there because it came later; 2.11BSD is actually still currently maintained, but still roughly resembles the system that was released in 1992 as the final backport of BSD features from 4BSD that would fit in the rather limited address space of the PDP-11). What you definitely won't find, however, is the actual challenge files, because they're on the tape image.

I put them on a tape image for two reasons. First, because I wanted to give a taste of how file transfer worked back in the days before networking was common and ubiquitous. Second, I wanted to give people a chance to solve it fully statically without running the system if they wanted, and it's much easier to extract from a tape image than from the disk image (and, correspondingly, much easier to get it into the emulated system as a tape than as a raw file).

As for how to load the tape, this is why the config file encourages you to read the manual. Recall from reading the config file that the tape unit is ts (the emulation for the TS11, a very common tape controller for PDP-11 systems). The brief procedure is:

- Break into the SIMH monitor (Ctrl-E by default).
- Attach the tape to the drive.
- Return to the running machine.

It looks a little something like this:

```
#
Simulation stopped, PC: 004376 (MOV (SP)+,177776)
sim> attach ts0 forth.tap
sim> continue
```

```
# cat /dev/rmt12
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.
```

Now we have the README, but how do we get the next file? It helps to know a bit of how tapes work, both overall and with Unix. With traditional magtapes, files are stored as a series of “records” of fixed length (because originally tapes were made to more efficiently store a bunch of punch cards in sequence, and each record was one card; by the time of Unix blocks were often 512 bytes, though larger archives sometimes used larger records to save tape overhead). Each file is then separated by a “file marker” (usually a longer blanked-out segment) and the next file would begin after that.
In Unix, as with most operating systems that dealt with tape, any operation would just leave the tape where it was so you could get to the next file pretty easily. There’s a tool that still exists in Unix and still

works on modern tape drives like LTO drives called mt (mag tape) for moving forward and backward between files, rewinding the tape, etc., but unless you've done something else here, you probably won't need it (though its man page has some helpful details as well). Instead, all we need to do is dump out the tape again and it will just go until EOF, at which point it will be stopped right before the next file (though this is the last file on the tape).

```
# cat /dev/rmt12 > foo
# ls -l
total 5
-rw-r-----  1 root         4591 Aug  8 17:02 foo
```

We have a file that looks like it could be big enough. Let's try marking it executable and running it.

```
# chmod +x foo
# ./foo
./foo: syntax error at line 4: `(' unexpected
```

That doesn't seem to work. Fortunately, the file command dates back to at least this point in Unix history and can give us some useful information about the file.

```
# file foo
foo:    block compressed 12 bit code data
```

It's compressed, but how? The only compression that exists on this Unix is the legacy compress utility; 2.11BSD was released in 1991 (though this particular patch set is from 2004), and gzip wouldn't be released until about a year later. Compress archives traditionally use the capital Z suffix, and the utility tends to expect this, so let's try renaming it and uncompressing it.

```
# mv foo foo.Z
# uncompress foo.Z
# ls -l
total 7
-rwxr-x--x  1 root         6388 Aug  8 17:02 foo
# file foo
foo:    executable not stripped
```

That looks more encouraging. Let's try running it.

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


```

That's cryptic. Let's hit the enter key a few times.

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

 ok

 ok
```

That's what I expect from Forth as a response! Astute Forth aficionados will note that we are getting a bonus newline here; that's because traditionally Forth controls the terminal a little more closely and doesn't actually print the newline when you hit enter, leaving that for the "ok" response to do, but I did not have the time or inclination to write the terminal control stuff to make that happen, so I just let the terminal driver handle that for me at the expense of behavior that's not 100% correct where it doesn't matter.
Typically, when you come to a new Forth system, it's helpful to use the word words to find out what the available commands (words) are. Let's try that.

```
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
```

It's a small subset (really mostly only what was needed to write the challenge) with a few interesting standouts. The first one, in particular, looks pretty juicy. Let's have a look at it.

```
secret
 ok
```

How mysterious! What did it do? We don't know. Let's see if it put something on the stack.

```
.
2728  ok
```

Okay, that's a number. Doesn't seem too useful on its own. Was that the only thing it put on the stack? We can find out with the word sp@, which is a pseudo-standard Forth word for getting the stack pointer.

```
sp@
 ok
.
3802  ok
```

We can see that after getting the secret and then popping it while printing the value, we returned to the same stack address as we started with. That number doesn't look like it's going to hold a flag all on its own, though, so maybe it's a pointer? Let's see what we get if we dereference it.

```
secret @ .
46  ok
```

We got a smaller number, which doesn't exactly help. Now, one thing to know about Forth is that there are two different common string representations: regular strings (represented by an address/count pair on the stack) and "counted" strings, represented by a pointer to a string prefixed with a length (a la Pascal, though the length doesn't have to be character sized). 46 seems like it could be the flag length (if a bit long). We can convert from counted strings to regular strings with the word count:

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

This doesn't mean a lot on its own, though, because count will happily convert anything you want to a split string representation, whether or not it is a string, because it's just pulling the first integer from the pointer. Let's try to print it.

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

That's obviously not it, but nothing blew up, so maybe there's something to this. We need to dig deeper than we can with just the executable, and for that, we need to use some tools.

More tools

This challenge can be done with external tools if they support PDP-11 disassembly and a.out binary parsing, but where's the fun in that? Let's live off the land a bit and use the tools that come with the system. There are two: nm (the name list tool) and adb (the debugger).

Getting the names with nm

The name tool (nm) is actually generally still around, though it is generally superseded by objdump (which is a GNU Binutils tool, so it does not exist here). It dumps the symbol table of a given executable, giving you the names of the symbols and their addresses and flags. Let's see what it comes up with for our sample.

```
# nm foo
000644 t _bugoff
000656 t _const
005332 b _data_rgn
007332 b _data_s0
005230 d _digits
000650 t _docol
011532 B _end
005330 d _here
000652 t _immed
005170 t _last
005250 d _secret
001702 t _set_cond
007332 b _tib_rgn
007532 b _user_rgn
005220 d _var_base
005224 d _var_holdend
005222 d _var_holdptr
005226 d _var_lastword
005216 d _var_ntib
005214 d _var_toin
001126 t _xor
001114 t _xor_xt
002266 t abort
002252 t abort_xt
000550 t accept
000534 t accept_xt
…elided for brevity…
002626 t word
002614 t word_xt
004420 t words
004404 t words_xt
001250 t zero
001236 t zero_xt
001676 t zeroeq
001666 t zeroeq_xt
002036 t zerogt
002026 t zerogt_xt
002022 t zerolt
002012 t zerolt_xt
002054 t zerone
002042 t zerone_xt
#
```

This is useful: we can recognize a lot of the words here (as well as equivalent names, and the _xt suffixes for same which you don't need to worry about because they're just part of the engine internals (but if you're curious, they're the "execution token" addresses for each word).
Of course, knowing the names doesn't really get us very far, maybe we should try to disassemble some things. Unfortunately, as far as I know there's nothing on this system that will just disassemble a file like objdump -D would, but there is a debugger called adb (not to be confused with the Android debugger which is also called adb, but is decidedly different) which will disassemble things if we point it in the right places. Let's try that.

Digging deeper with adb

Let's pry a little deeper with adb, the original Unix debugger. This syntax will likely be fairly alien to you if you're used to GDB or WinDbg, etc. It's a lot more akin to the syntax you might expect from debug capabilities in a bootloader, which makes sense given the limited resource footprint it had to fit into. If you read the man page as the second README suggested, you'll note that there's a tutorial document mentioned at the end which is still pretty easy to find online, written by one S.R. Bourne (yes, that's the Stephen Bourne of Bourne shell fame). It helps a lot with understanding the syntax.
Similar to Forth, the operands go first and then comes the operator, sometimes followed by flags. Thus, to set a breakpoint, we would say <address>:b where <address> can be numeric or symbolic. Let's try that to see what happens when we execute the word secret.

```
adb> # adb foo
adb> secret:b
adb> :r
foo: running
MoogleForth starting. Stack: 3802
secret
breakpoint      secret:         jsr     pc,_const
adb> _const?
_const:         013645  = mov   *(sp)+,-(r5)
adb>
_const+02:      012407  = mov   (r4)+,pc
```

When we hit the word "secret", it immediately jumps to a routine called _const. What does that do? We can find out with the question mark operator, which examines the contents of an address (the default is to try to disassemble it, which is valid in a lot of cases, but it also shows the numeric value). It looks like _const pulls the target of the stack pointer and dereferences that, then increments it, storing the result in the target of a pre-decremented r5 (note that because this is Unix, this is AT&T syntax which has destinations on the right). It then stores the target of r4 into pc (effectively performing a jump) and increments r4.
So what does this mean? The top of stack after a jsr pc,<target> instruction is going to be the return address (the jsr instruction can also take operands other than pc, in which case it uses that register as a link register instead of pushing the next address directly on the stack). Thus, the first instruction here takes that return address (which is immediately after the jsr instruction), dereferences it to get the contents, and increments it (by 2 because this is a word-sized operation). The resulting value is moved into the pre-decremented r5 location, which we can assume is the operation of pushing it onto the Forth data stack.
The next instruction pulls the contents of r4 and jumps to it, incrementing r4 in the process. This is because this implementation of Forth, like many others, is "threaded", meaning the Forth "program" is a list of addresses of functions, and r4 is effectively the Forth instruction pointer. This instruction thus executes the next Forth word in the program, similar to a return.
From this, we can conclude that the _const function loads whatever is after the jsr instruction onto the Forth data stack, effectively loading a constant. Why not just have secret load it directly onto the stack? Part of the reason is because this makes it much easier later when implementing the see word (essentially a Forth disassembler) because you can identify the fixed jsr pc,_const pattern and print

the function as a constant instead of as literal assembly. There's a similar pattern with actual Forth words that aren't native code, as we'll see in a moment.
Anyway, what does this mean for us? It means that secret just loads a constant, whose value is:

```
adb> secret?
secret:         04767   = jsr   pc,_const
adb>
secret+04:      05250   = inc   *-(r0)
```

It's 05250, which the zero prefix should tell you is octal (a lot of DEC stuff still used octal when Unix was created because all PDP machines before the 11 were 18, 12 and 36 bit, which made octal more practical than hex, plus it's simpler to decode from ASCII because there's no alphabet gap). In decimal, that's 2728, which you can see is the value we wind up with on the stack when we execute secret. Similarly, if we look inside the address (this time in decimal), we find the same first word:

```
adb> 05250?d
lastword+042:   46
```

You could solve this without digging much further if you made the mental leap about this being an encoded, encrypted counted string and did a bit of cryptography. If you assume this is the flag value, you could dump the bytes and start chewing on it (it's not a particularly difficult encoding or cipher, but might not be the easiest to guess). But it's more fun to figure out how it actually ticks.

Decoding

Let's have a look at decode, which also looked pretty interesting.

```
adb> decode?
decode:         jsr     r4,_docol
```

This one looks pretty similar to the constant, with two fairly important differences. First, it's jumping to a routine called _docol, which is the traditional Forth name for "do colon" because Forth words are called "colon definitions" since the colon is the word you use to compile the word. This implementation doesn't have a compiler or a colon word, but you'll see docol referenced in lots of Forth literature.

The second difference is that instead of pc, the first operand is r4 (our instruction pointer). Recall from earlier that in the versions of jsr that don't use pc as the first operand, the provided register is used as the linkage register (storing the return address that would normally go directly on the stack) and the linkage register is pushed on the stack (when you think of it, this isn't really a special case because it's still true with pc, it's just that the pc gets overwritten immediately after by the branch target). This is useful for us to maintain the separate Forth instruction pointer. The next address after the jsr instruction goes into the Forth instruction pointer, and the old Forth instruction pointer gets stacked so it can be returned to later. Let's look inside _docol.

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

It's just the one instruction, actually the same as the last instruction we saw in _const. It loads the next address from the Forth instruction pointer into the pc, jumping to it. Recall that r4 was just loaded as the next address after the jsr instruction, so the list of Forth words should follow after that. Let's look at those. We'll use adb's p format to list them symbolically, since they're pointers to code to which symbols already point.

```
decode:         jsr     r4,_docol
adb>
decode+04:      bne     0177754
adb> .?p
decode+04:      zero
adb>
decode+06:      tor
adb>
decode+010:     dup
adb>
decode+012:     if
adb>
decode+014:     decode+052
adb>
decode+016:     swap
adb>
decode+020:     dup
adb>
decode+022:     cat
adb>
decode+024:     rto
adb>
decode+026:     plus
adb>
decode+030:     twodup
adb>
decode+032:     swap
adb>
decode+034:     cbang
adb>
decode+036:     tor
adb>
decode+040:     oneplus
adb>
decode+042:     swap
adb>
decode+044:     oneminus
adb>
decode+046:     goto
adb>
decode+050:     decode+010
adb>
decode+052:     rdrop
adb>
decode+054:     twodrop
adb>
decode+056:     exit
adb>
decrypt_xt:     decode_xt
adb>
```

Note that the initial jsr is two words long because the offset takes up a full word. After that, we start listing the addresses symbolically until we move past the end of decode. The last word is exit. The Forth interpreter will just keep going down that list, executing the contents of those addresses until it's redirected. What does exit look like?

```
adb> exit?i
exit:           mov     (sp)+,r4
adb>
exit+02:        mov     (r4)+,pc
```

This pops the top of stack (which should be the stacked previous Forth instruction pointer unless something has gone horribly wrong) into r4, and then does the familiar “next” operation. We have to do this one-two punch to make sure the old r4 gets loaded properly.
In the decoding listing you see mostly some fairly intuitive names along with some cryptic ones, most of which are due to symbol name limitations, e.g. “rto” is the word r> and “tor” is >r, which move the top of the data stack to and from the return stack, respectively (this is something you see frequently in some lower-level Forth words to store values temporarily without screwing up the data stack). Condensed and translated, this would look something like this (this is more or less what “see” would have shown if I’d had time to implement it):

```
decode+04:      0
decode+06:      >r
decode+010:     dup
decode+012:     if
decode+014:     decode+052
decode+016:     swap
decode+020:     dup
decode+022:     cat
decode+024:     r>
decode+026:     +
decode+030:     2dup
decode+032:     swap
decode+034:     c!
decode+036:     >r
decode+040:     1+
decode+042:     swap
decode+044:     1-
decode+046:     goto
decode+050:     decode+010
decode+052:     rdrop
decode+054:     2drop
```

I’m not going to delve into the specifics of these except to say that the control words just take the next value and jump to it, sometimes conditionally. Real Forth engines have an entire control stack concept in the compiler which I did not implement here, so they have fancier concepts like actual loops instead of just gotos, but they essentially compile down to this as well. We have one problem: we don’t know what kind of input this expects. I’ll expand this with Forth stack notation and substitute in placeholders like ?1, ?2, etc. when we come across previously unknown input stack values, with any return stack values in parentheses.

```
decode+04:      0               ; 0
decode+06:      >r              ; (0)
decode+010:     dup             ; (0) ?1
decode+012:     if              ; (0)           ; If top of stack is zero, do the jump
over
decode+014:     decode+052                      ; Conditional jump address
decode+016:     swap            ; (0)           ; This swaps two input stack locations
decode+020:     dup             ; (0) ?2        ; So now ?2 is on top
decode+022:     c@              ; (0) *?2       ; Get the character from ?2
decode+024:     r>              ; *?2 0         ; Pop the return stack
decode+026:     +               ; *?2+          ; Add... zero?
decode+030:     2dup            ; *?2+ ?2 *?2+  ; Copy that value plus the original
pointer
decode+032:     swap            ; *?2+ *?2+ ?2  ; Swap them...
decode+034:     c!              ; *?2+          ; So that we can store the result in ?2
decode+036:     >r              ; (*?2+)        ; Put that back on the return stack
decode+040:     1+              ; (*?2+)        ; Increment ?2 (hidden)
decode+042:     swap            ; (*?2+)        ; Swap ?1 and ?2 back to their original
positions
decode+044:     1-              ; (*?2+)        ; Decrement ?1
decode+046:     goto            ; (*?2+)        ; Loop back to decode+010
decode+050:     decode+010
decode+052:     rdrop           ;               ; Clean up the return stack
decode+054:     2drop           ;               ; And drop our original inputs
```

Doing a brief analysis of this, we can see that there are two input variables. ?1 looks like it's a counter, and ?2 looks like a character address, so it seems clear that this operates on split strings. If we refine the listing with that knowledge, it looks like:

```
                                ; ca cn         ; Input values (char address and count)
decode+04:      0               ; ca cn 0
decode+06:      >r              ; (0) ca cn
decode+010:     dup             ; (0) ca cn cn
decode+012:     if              ; (0) ca cn     ; If count is zero, skip to the end
decode+014:     decode+052                      ; Conditional jump address
decode+016:     swap            ; (0) cn ca     ; Swap address and count
decode+020:     dup             ; (0) cn ca ca  ;
decode+022:     c@              ; (0) cn ca c   ; Get the character from ca
decode+024:     r>              ; cn ca c 0     ; Pop the return stack
decode+026:     +               ; cn ca c+      ; Add... zero?
decode+030:     2dup            ; cn ca c+ ca c+ ; Copy that value plus the original
pointer
decode+032:     swap            ; cn ca c+ c+ ca ; Swap them...
decode+034:     c!              ; cn ca c+      ; So that we can store the result in ca
decode+036:     >r              ; (c+) cn ca    ; Put the decoded char on the return
stack
decode+040:     1+              ; (c+) cn ca+   ; Increment the string pointer
decode+042:     swap            ; (c+) ca+ cn   ; Swap ca/cn back to their original
positions
decode+044:     1-              ; (c+) ca+ cn-  ; Decrement ?1
decode+046:     goto            ; (c+) ca+ cn-  ; Loop back to decode+010 with updated
values
decode+050:     decode+010
decode+052:     rdrop           ; ca+ cn-       ; Clean up the return stack
decode+054:     2drop           ;               ; And drop our original inputs
```

That's a little bit cleaner. You can see that the value that goes in and out of the return stack starts at zero but holds the value of the last decoded character, and is then added to the next input character

to get the next decoded value and so forth; this is thus a simple delta encoding to trivially obfuscate the string to avoid casual known plaintext attacks against the flag (it's still not much of an obstacle for a good cryptographer, but I didn't want to make it too easy). The end result is a clean stack with the value modified in place.

Okay, so how do we get the split string from the counted string? We could do it manually, or we could use the standard Forth word count, which does it for us (as shown earlier). That still results in something illegible, because we haven't decrypted it yet, but at least it doesn't seem to blow up.

```
# ./foo
MoogleForth starting. Stack: 3802
secret count 2dup decode type
?hZVrg
 akY_+WXEuwmDxP0w`b;J ok
```

I guess now we need to look at decrypt.

Decrypt

Let's see the first instruction.

```
adb> decrypt?
decrypt:        010246  = mov   r2,-(sp)
```

That's interesting, there's no call to _const or _docol or really anything at all. This is a natively-implemented word, so it just runs as code. Let's take a look at it (I'm omitting the intervening lines for brevity):

```
adb> decrypt?
decrypt:        010246  = mov   r2,-(sp)
decrypt+02:     010346  = mov   r3,-(sp)
decrypt+04:     010446  = mov   r4,-(sp)
decrypt+06:     010546  = mov   r5,-(sp)
decrypt+010:    04467   = jsr   r4,bl
decrypt+014:    0342    = swab  -(r2)
decrypt+016:    012604  = mov   (sp)+,r4
decrypt+020:    04467   = jsr   r4,parse
decrypt+024:    0352    = swab  *-(r2)
decrypt+026:    012604  = mov   (sp)+,r4
decrypt+030:    012503  = mov   (r5)+,r3
decrypt+032:    012501  = mov   (r5)+,r1
decrypt+034:    011502  = mov   (r5),r2
decrypt+036:    016500  = mov   02(r5),r0
decrypt+042:    010346  = mov   r3,-(sp)
decrypt+044:    020203  = cmp   r2,r3
decrypt+046:    03002   = bgt   decrypt+054
decrypt+050:    010203  = mov   r2,r3
decrypt+052:    01411   = beq   decrypt+076
decrypt+054:    0160302 = sub   r3,r2
decrypt+056:    0111004 = movb  (r0),r4
decrypt+060:    0112105 = movb  (r1)+,r5
decrypt+062:    074405  = xor   r4,r5
decrypt+064:    0110520 = movb  r5,(r0)+
decrypt+066:    077305  = sob   r3,decrypt+056
decrypt+070:    011603  = mov   (sp),r3
decrypt+072:    0160301 = sub   r3,r1
decrypt+074:    0763    = br    decrypt+044
decrypt+076:    05726   = tst   (sp)+
decrypt+0100:   012605  = mov   (sp)+,r5
decrypt+0102:   012604  = mov   (sp)+,r4
decrypt+0104:   012603  = mov   (sp)+,r3
decrypt+0106:   012602  = mov   (sp)+,r2
decrypt+0110:   012407  = mov   (r4)+,pc
emit_xt:        0306    = swab  sp
```

You can see that this ends with the now-familiar mov (r4)+,pc sequence that serves as the Forth "next" instruction; essentially, this code executes like any other Forth word (which means if we call out to it, we need to call it like we would a Forth word instead of just a regular function or it won't return properly).

Let's look at the code a bit. It stacks a bunch of registers first (if you look elsewhere in the native code, r0 and r1 are treated as volatile registers, while everything else needs to be preserved). We can see them unstacked at the end as well. Immediately after that, we have this curious sequence:

```
decrypt+010:    04467   = jsr   r4,bl
decrypt+014:    0342    = swab  -(r2)
decrypt+016:    012604  = mov   (sp)+,r4
```

Here we call the standard bl word, which just deposits an ASCII space ("blank") on the stack, but it's followed by something that doesn't make a whole lot of sense. Why swap the decremented contents of r2, which should be treated as uninitialized? It's because that's not actually an instruction.

```
adb> decrypt+016=
                0342
```

Recall that the “next” sequence is mov (r4)+,pc and that jsr r4,<target> stacks the old value of r4 and puts the next pc value into r4. When we call a Forth word from native code, we need to provide it with some “breadcrumbs” to come back to the native code or else it’ll go off to wherever r4 is pointing, which is probably not where we really wanted to be. So we make the next value a pointer to the next instruction, which is a little goofy, but it gets the job done. The next instruction then does the job of popping the old r4 value off the return stack so when we “next”, we go to the original “next” when this function was called.

There was probably a slightly more elegant way of doing this with fancy return modes, but I didn’t spend too much time on it in favor of just getting things working.

So now that we know about that calling sequence, we can refine this a little bit as if it were macros (which is exactly what it is in my assembler file):

```
decrypt:        010246  = mov   r2,-(sp)
decrypt+02:     010346  = mov   r3,-(sp)
decrypt+04:     010446  = mov   r4,-(sp)
decrypt+06:     010546  = mov   r5,-(sp)          ; Stack the register values
decrypt+010:    04467   = _CALL(bl)               ; Call bl while we still have the data stack
decrypt+020:    04467   = _CALL(parse)            ; Call parse (which returns a split string)
decrypt+030:    012503  = mov   (r5)+,r3          ; Pop the string count into r3
decrypt+032:    012501  = mov   (r5)+,r1          ; Pop the address into r1
decrypt+034:    011502  = mov   (r5),r2           ; Copy the next top of stack to r2
decrypt+036:    016500  = mov   02(r5),r0         ; And what was under it to r0
decrypt+042:    010346  = mov   r3,-(sp)          ; Push the string count to the other stack
decrypt+044:    020203  = cmp   r2,r3             ; Compare with what’s in r2 (the input count?)
decrypt+046:    03002   = bgt   decrypt+054       ; If r2 was greater, skip this next bit
decrypt+050:    010203  = mov   r2,r3             ; If not, copy r2 to r3 (r3=min(r2,r3))
decrypt+052:    01411   = beq   decrypt+076       ; If the result was zero, skip to the end
decrypt+054:    0160302 = sub   r3,r2             ; Either way, subtract our input counts
decrypt+056:    0111004 = movb  (r0),r4           ; Move a char from (r0) to r4
decrypt+060:    0112105 = movb  (r1)+,r5          ; And a char from our parse buffer++
decrypt+062:    074405  = xor   r4,r5             ; XOR them together into r5
decrypt+064:    0110520 = movb  r5,(r0)+          ; Store the result back into (r0), increment
decrypt+066:    077305  = sob   r3,decrypt+056    ; Decrement r3 and loop back if nonzero
decrypt+070:    011603  = mov   (sp),r3           ; If it hit zero, reload it from the stack
decrypt+072:    0160301 = sub   r3,r1             ; And use it to reset the input pointer
decrypt+074:    0763    = br    decrypt+044       ; Loop back to the r3=min(r2,r3) operation
decrypt+076:    05726   = tst   (sp)+             ; Drop the temp value on the return stack
decrypt+0100:   012605  = mov   (sp)+,r5          ; Restore the data stack and everything else
decrypt+0102:   012604  = mov   (sp)+,r4
decrypt+0104:   012603  = mov   (sp)+,r3
decrypt+0106:   012602  = mov   (sp)+,r2
decrypt+0110:   012407  = _NEXT                   ; Go to the next word
```

This one needs you to know how the word parse works. Fortunately, that’s also a standard Forth word, and it’s implemented according to the standard. I’m not going to delve into the specifics of it (but you can look if you want), but it basically takes the desired delimiter as an input parameter and grabs the next word from the interpreter input and returns it on the stack as a split string (it is part of the core of the interpreter REPL loop). Here, we feed it a space, much like the regular parser uses.

Looking through the code, the result gets transferred to r3 (count) and r1 (pointer). Then the function also transfers the next two parameters from the stack to local registers; we can infer from the functionality that this is also a string, with the count stored in r2 and the address stored in r0. The

remainder of the function steps through and performs an in-place keyed XOR using the string grabbed from parse as the key. The pseudocode looks a lot like this:

```
Stash r2-r5 on the stack (this includes the original data stack pointer)

Call parse with an ASCII space as the parameter

r3 = keylen
r1 = keybuf
r2 = msglen
r0 = msgbuf
saved_len = keylen

loop:
  r3 = min(r2,r3)        // Get the min of the key len or remaining msg len
  if(r2 == 0): break     // Bug out if wwe're at the end

  r2 -= r3             // Subtract our inner loop length from the remaining msg len
  while(r3--):         // Loop over our inner loop length
    *(msgbuf++) ^= *(keybuf++) // XOR the values
  r3 = saved_len       // Restore the key length
  r1 -= r3             // Use it to reset the key buffer pointer

Restore the stashed r2-r5 (including the original data stack pointer)
```

So this takes the key from parse and uses it to decrypt the value passed in as input parameters. Unlike decode, this one returns the original split string pointer/length on the stack (though the value inside has been decoded). Real Forth words often are inconsistent like this, and I wanted to give a flavor of all the passing/return methods.

Remember how I said you'd need Ken Thompson's password? This is where that comes into play. Ken's password (among others) was cracked a few years ago from the encrypted value present in early Unix distribution passwd files. I chose this one because it was hard to guess (he reportedly couldn't remember it until it was revealed, at which point he recalled it was a chess move), fit the pattern suitable for Forth words with the standard parser, and was related to both Google and ancient Unix (and is pretty easy to find currently). Ken Thompson's password was: `p/q2-q4!`

Wrapping it up

In summary:

- We can get the encoded, encrypted secret from the word secret as a counted string
- We can convert that from a counted string to a split string using the word count
- We can decode the resulting string with the word decode, which does not return anything
- We can pass the decoded split string to the word decrypt and supply it with the password in the input buffer, and it will return the decrypted string
- We can print the string with the word type

You may have noticed that there's one thing we don't know: which order do decode and decrypt have to be in? There's no way to tell from the code, and the order does matter because of the adds in decode, so from here we'll have to guess. Let's try it in the order it's been presented:

```
# ./foo
MoogleForth starting. Stack: 3802
secret count 2dup decode decrypt p/q2-q4! type
8a\*xN.t.v(*pw6~zTB5J}ACAbMb(; ok
```

Well, that's not right. Let's try the reverse.

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

Or, for slightly more intuitive 2dup placement:

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

That looks like a standard-format flag! We're done here. I hope you enjoyed running through this as much as I enjoyed writing it!