MANDIANT®
NOW PART OF Google Cloud

FLARE-ON CHALLENGE 9 SOLUTION

BY DAVE RILEY

Challenge 10: Nur geträumt

Challenge Prompt

This challenge is a Macintosh disk image (Disk Copy 4.2 format, for those who need to know) containing a 68K Macintosh program. You must determine the passphrase used to decode the flag contained within the application. Super ResEdit, an augmented version of Apple's ResEdit resource editor which adds a disassembler, is also included on the disk image to help you complete the challenge, though you will likely also need to do some outside research to guess the passphrase.

This application can be run on any Macintosh emulator (or any real Macintosh from as far back as a Mac Plus running System 6.0.x up to a G5 running Classic). The setup of the emulation environment is part of the challenge, so few spoilers live here, but if you want to save yourself some headaches, Mini vMac is a pretty good choice that doesn't take much effort to get up and running compared to some other options.

This application was written on a Power Macintosh 7300 using CodeWarrior Pro 5, ResEdit, and Resourcerer (my old setup from roughly 1997, still alive!). It was tested on a great many machines and emulators and validated to run well on Mac OS from 6.0.8 through 10.4.

Happy solving! Be curious!

7-zip password: flare

Solution

This challenge has a few different ways to solve it, some of which do not even require reverse engineering (though it certainly helps). Indeed, you may derive as much benefit from being a GenX-er or knowing some German as you do from being a skilled reverse engineer (and in any case, most likely you need to brush up on your 68K assembly and/or German skills anyway).

This challenge was derived from a previous version whose traces you may have been able to find in your searches online for other clues, but I have upgraded the difficulty somewhat (and fixed some bugs) so that the same solution path won't exactly work.

The Environment

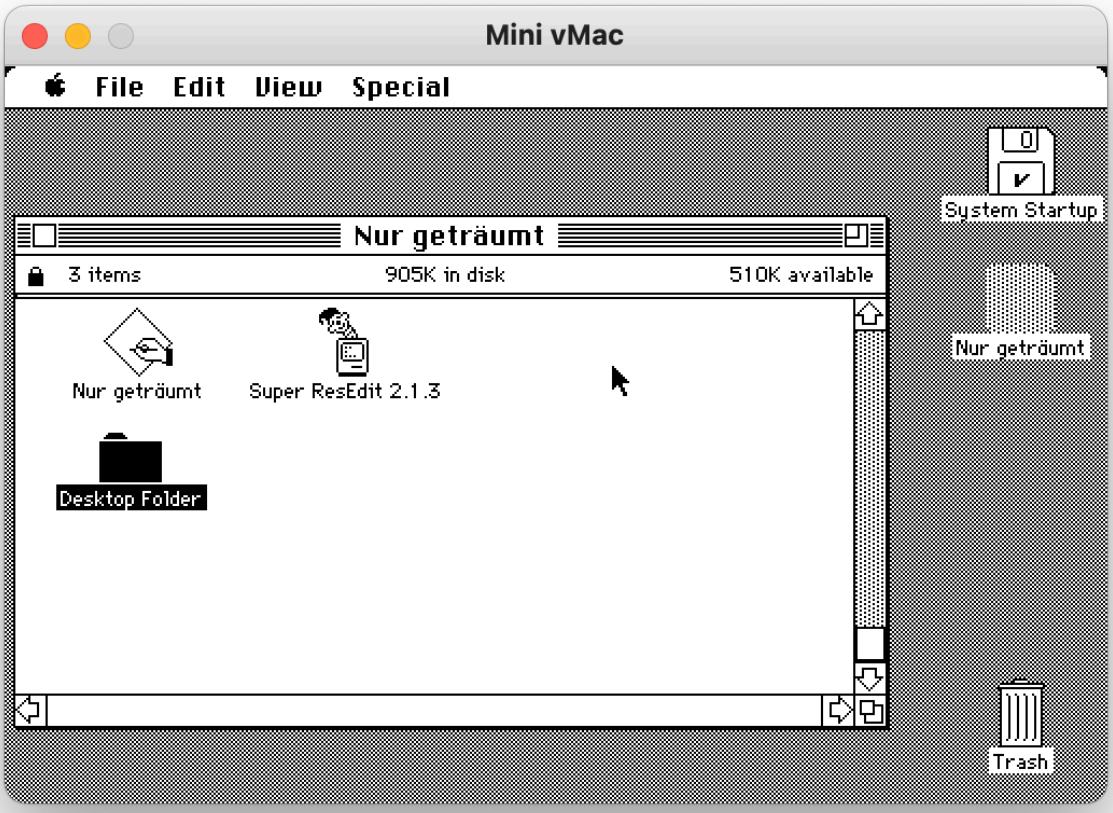
The first challenge, of course, is getting the environment up and running. This challenge **does** run just fine on a real vintage Mac (anything from a Mac Plus running System 6.0.7 up to a PowerPC Mac running Mac OS X 10.4, if you have the Classic environment installed), just like it says on the box. Go ahead, try it if you can! I can even send you a real floppy with the challenge on it if your Mac can take high-density disks.

Everyone else, though, needs to run on an emulator. It's been tested on those, too, of course. As mentioned in the prompt, Mini vMac is the easiest out-of-the-box solution (compared to, say, Basilisk II), but even that requires some exploring. For one thing, you need to get a ROM file for any Mac emulator to work, which means you obviously borrowed your friend's old Mac so you could legitimately claim possession of a ROM which you downloaded from one of the many Mac abandonware sites, right? Right. That will get you to booting, but you still don't have an operating system.

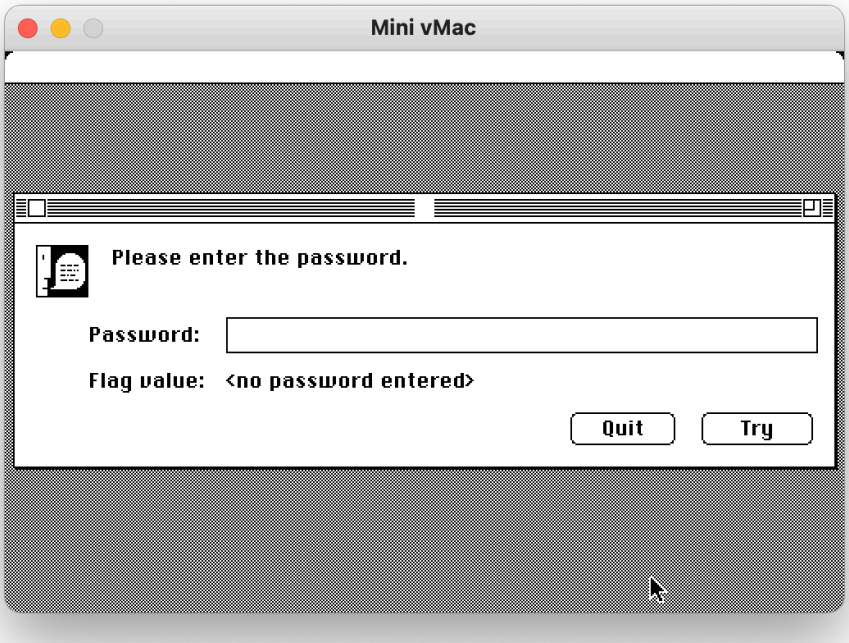
Fortunately, Apple did release several versions of its operating systems for free, though the license under which they were released doesn't permit redistribution of installed versions (as far as I understand it), so we couldn't distribute boot media. Fortunately, you can create your own boot media by making a new disk image and running the installer disk images on it (System 6.0.8 will be your easiest bet here, as none of the System 7 images are mountable/bootable by emulators). Or you might have found an unauthorized pre-made boot disk image online while you were searching! We obviously don't condone this, but there are a lot of them out there.

The Program

Now you have a bootable disk, and you've mounted the challenge disk image in your emulator. You're ready to start on the challenge!



Let's open the application. Notice that this is a German name, with an umlaut! That will be a useful clue for the solution (along with the name itself).



Well, that's a little cryptic, isn't it? Let's put in a test password.

That looks like a lot of garbage. The first few characters look like Roman characters, but it's clearly not the flag (which needs to end with @flare-on.com). Maybe now is a good time to explore the other application that comes with it, Super ResEdit.

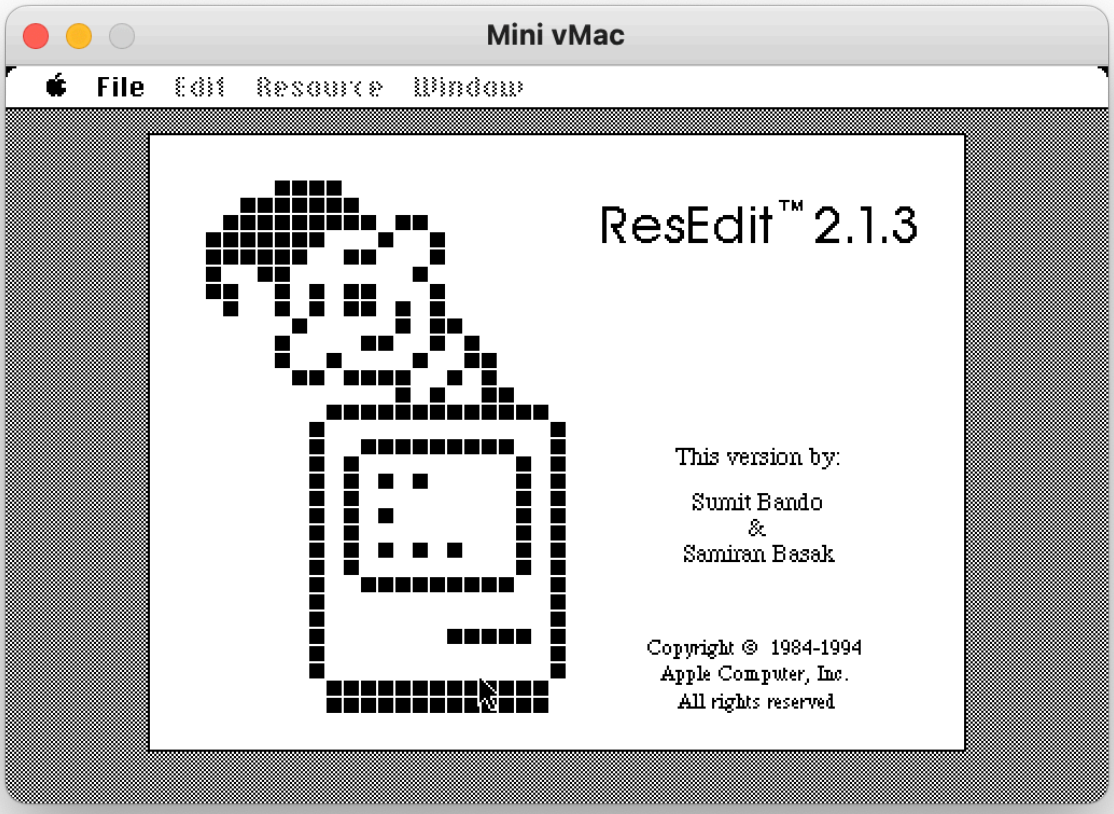
Super ResEdit

A bit of background: from the beginning, the Macintosh had the concept of "resources", which were a part of a file in a separate "fork" (Windows NT and POSIX operating systems have similar concepts to forks, namely Alternate Streams and Extended Attributes, respectively) which had a defined format for naming and manipulating individual resources with a framework called the Resource Manager. Windows adopted this idea for the Resources section of PE files later and used it for a lot of the same purposes (defining windows, menus, icons, etc.), though the implementation differs somewhat from the Mac approach. All Macintosh files have a resource fork and a data fork, though each may be empty (for example, text files use the data fork exclusively as a stream of bytes, just like any normal file in other operating systems).

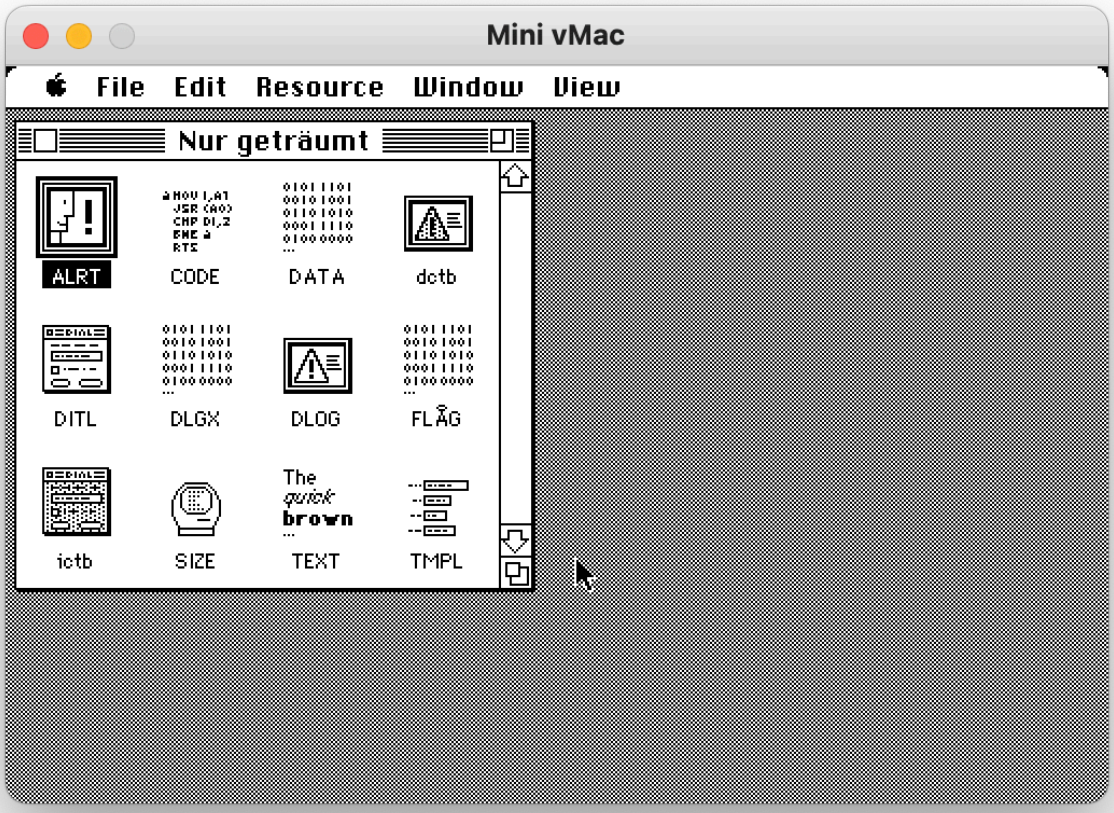
ResEdit was Apple's free developer tool for manipulating resources in files. It was extremely handy for a lot of things beyond just development. Super ResEdit is a version of ResEdit patched with a few additional handy things, most notably here a 68k disassembler (also from Apple, just never officially distributed outside Apple).

Let's look at our application file in Super ResEdit.

If your boot disk is read-only, you will get this message. That's fine, you don't need to save the preferences anyway, probably.

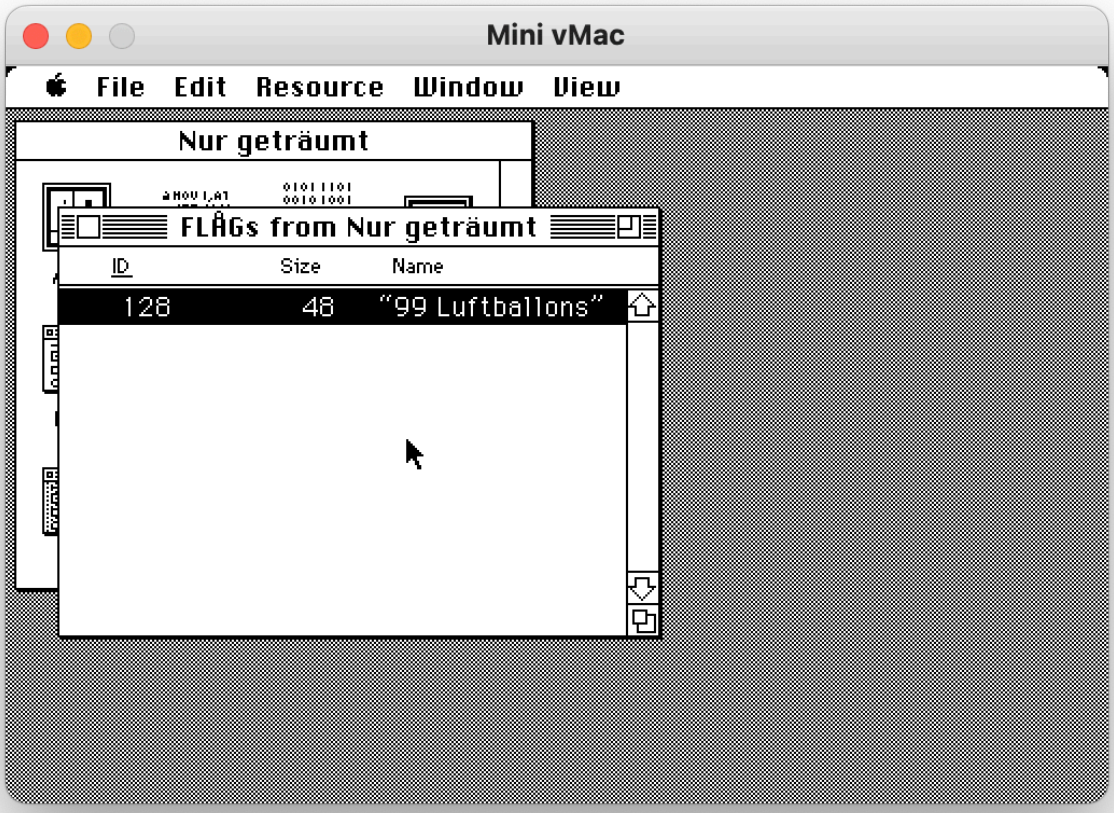


This is the splash screen, with an animated jack-in-the-box.

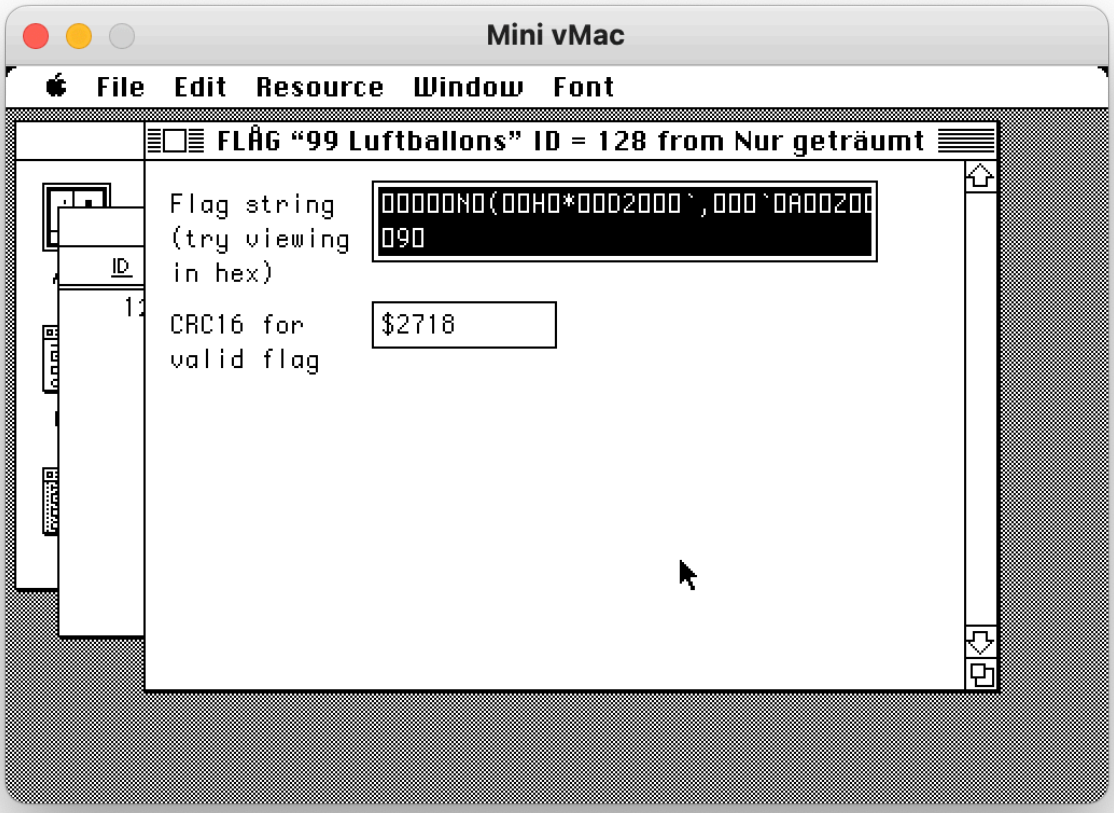


And when we open the application, this is what we see. Sixteen resource types! Some of these (the ones known to ResEdit) have their own normal icons, while others (custom resources, or ones from later versions of Mac OS like the DLGX resource) have a "ones-and-zeroes" icon to indicate that they're unknown data. Note: the DATA resource is just the entire resource fork as a flat piece of data, and it doesn't really exist, it's a Super ResEdit special.

I'm not going to go into the structure of the dialog resources here, they're just to make the GUI run (and they're not even needed, strictly speaking, but it's both more convenient and more compact to manipulate the dialogs as resources than define them in code). But maybe we should have a look at the FLÅG resource, hm?



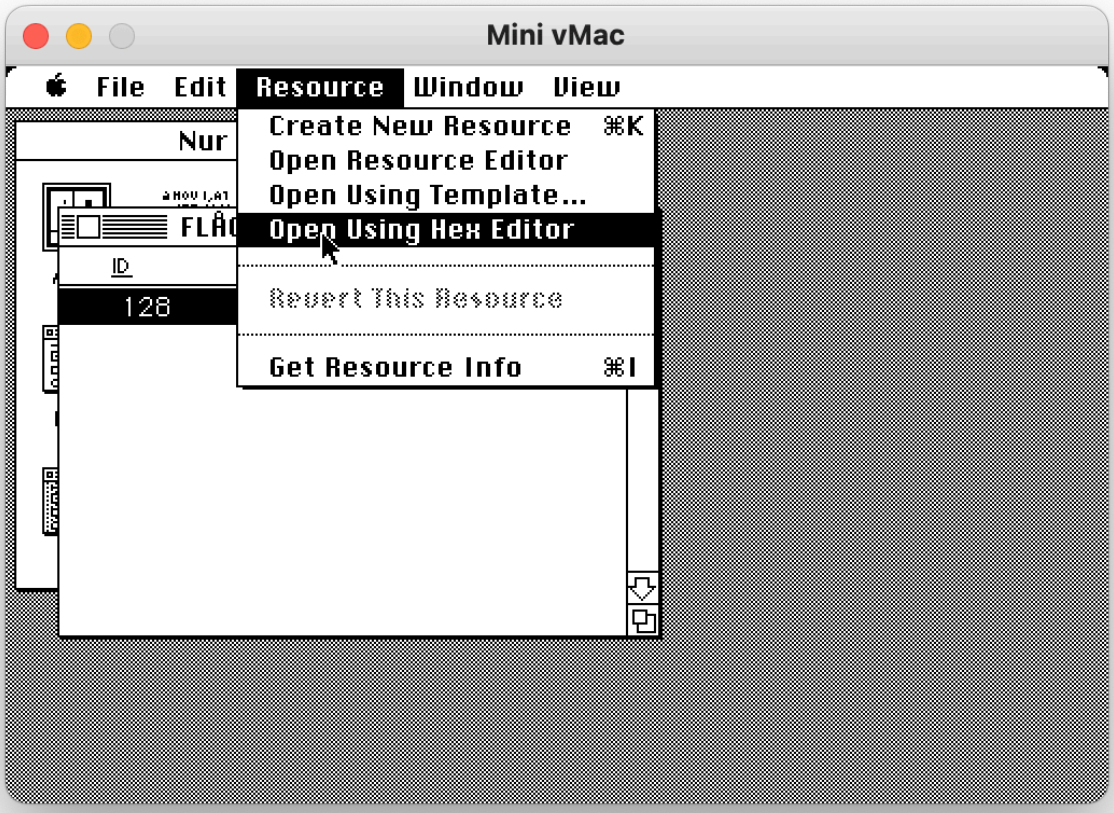
Okay, so there's one resource of this type, with ID 128 (resource IDs below 128, including all negative ones, are reserved for the system for Reasons, so they are created starting at 128 by default), named "99 Luftballons". You may be noticing a theme here; we'll discuss that later. For now, let's open that up.

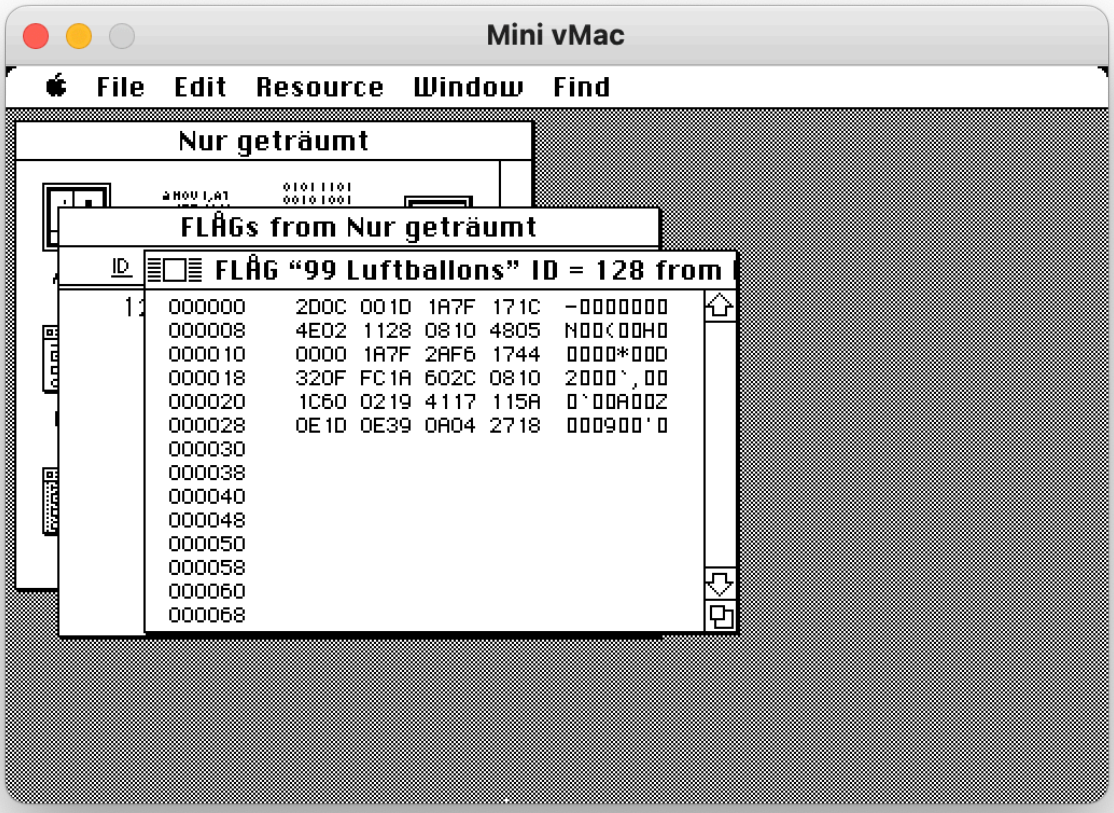


Hey, it's formatted! That's because there's a TMPL resource in this file for that resource type (go ahead and look if you'd like), which tells ResEdit how to format the data for human consumption. It's generally useful, though ResEdit doesn't have a template type for hex strings, or I would have used that. Thus, the template gives us the clue to look in hex.

Before we do that, though, note that there is also a CRC16 for the valid flag! This is how the program determines whether you've gotten the correct password, because the resulting flag will match the CRC. Don't try to brute-force it, though, because a) the password is much too long and b) it's in German and c) it's not even in a convenient encoding for much hash busters and d) given the rather small output space of CRC16, you're much more likely to wind up with a nonsense hash collision than the correct value. Don't do it.

Anyway, on to the hex value.

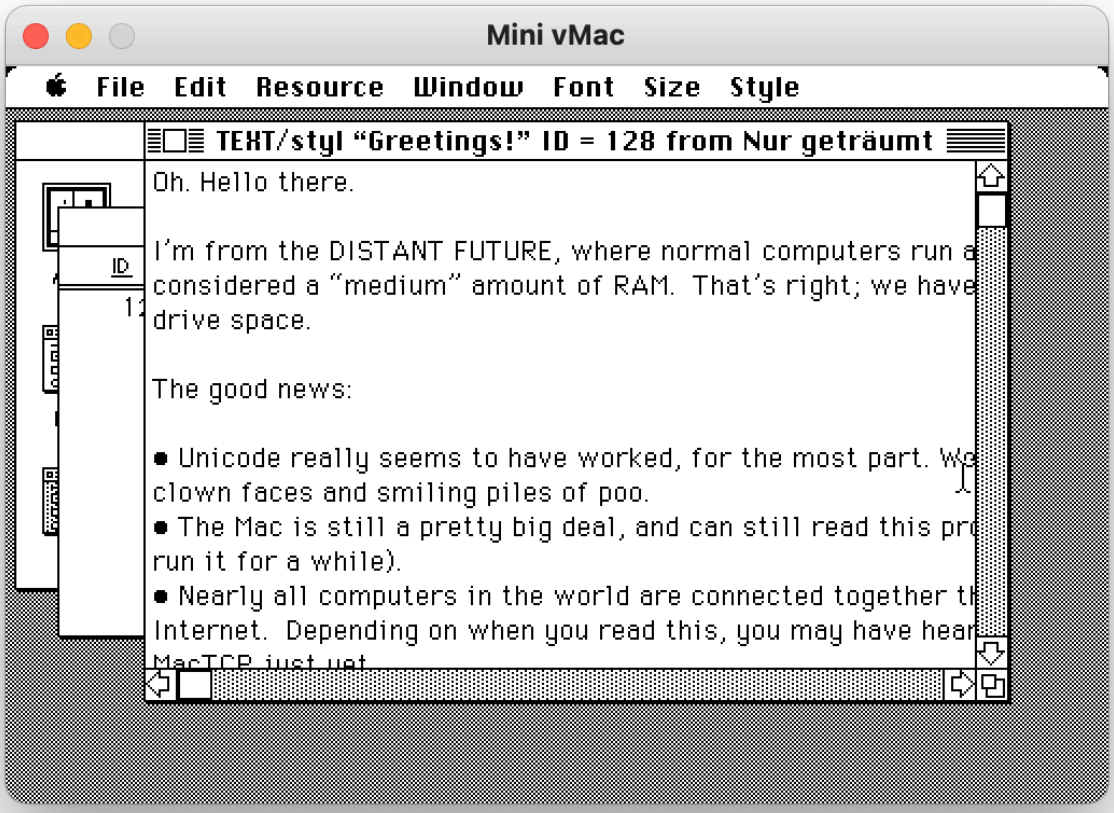

Mini vMac
File
Edit
Resource
Window
View
Create New Resource ⌘K
Open Resource Editor
Open Using Template...
Open Using Hex Editor
Revert This Resource
Get Resource Info ⌘I
Nur
ID
128



Okay, what do we make of this? The template defines packed binary data, so everything is just squeezed together with no delimiters here. We know the CRC16 should take up 2 bytes if we looked at the template or just guessed it, which means that the string itself should be the first 0x2E bytes. There's no NULL terminator, but: that's because this is a Pascal string, which has a length byte at the beginning instead. As it turns out, that length byte is 0x2D, which is perfect for our overall string length (the length byte itself does not count, so Pascal strings can be up to 255 characters long).

However, the flag itself doesn't yield a lot of useful data; it still looks like garbage on the ASCII side on the right (technically, it's MacRoman, not ASCII, which is a similar concept to the extended ASCII alphabets DOS and Windows use, but not the same encoding; this is somewhat important).

What else could we look at for clues? There's a TEXT resource that might be useful...



Here, I must apologize; I wrote this up on a larger screen without realizing that ResEdit doesn't wrap the text in the box to the window, and there didn't seem to be a way to set the margins. That makes this excruciating to read on the small screen presented by Mini vMac. Here is the text in its entirety:

```
Oh. Hello there.

I'm from the DISTANT FUTURE, where normal computers run at 2-4 GHz and 16 GB is considered a
"medium" amount of RAM.  That's right; we have more RAM than you have hard drive space.

The good news:

• Unicode really seems to have worked, for the most part. We even have characters for clown
faces and smiling piles of poo.
• The Mac is still a pretty big deal, and can still read this program (but hasn't been able to
run it for a while).
• Nearly all computers in the world are connected together through a network called the
Internet.  Depending on when you read this, you may have heard of it. Don't get rid of MacTCP
just yet.

The bad news:

• Nearly all computers in the world are connected together through a network called the
Internet. This has made a lot of people very angry and been widely regarded as a bad move.
```

• Despite having 16 GB of RAM, Microsoft Word still takes up roughly half of it.
• We're still using Microsoft Word.

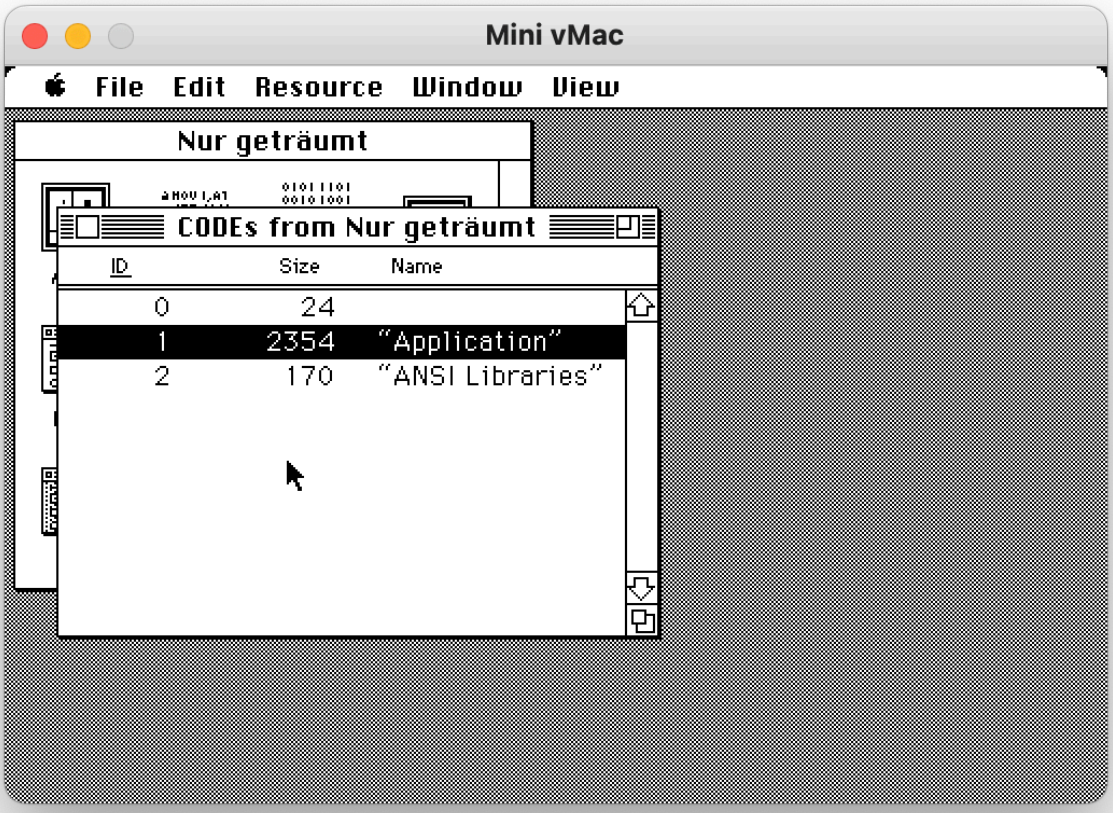
Anyway, because in the future we're stuck at home due to a worldwide pandemic (no, not the Internet, there is ANOTHER one), we had a competition for finding fun things in computer programs. I've hidden a flag in this program, but it's not going to be all that easy to find just with ResEdit.

You'll probably need to interact with the program a bit. It will let you know when you've found the right flag. I've done you the favor of including Super ResEdit here, which has a disassembler, and I'm even nice enough to give you the debug version of the program with all the symbol names, to give you a head start (because I'm not wicked enough to make you step through it with MacsBug, but you could if you wanted).

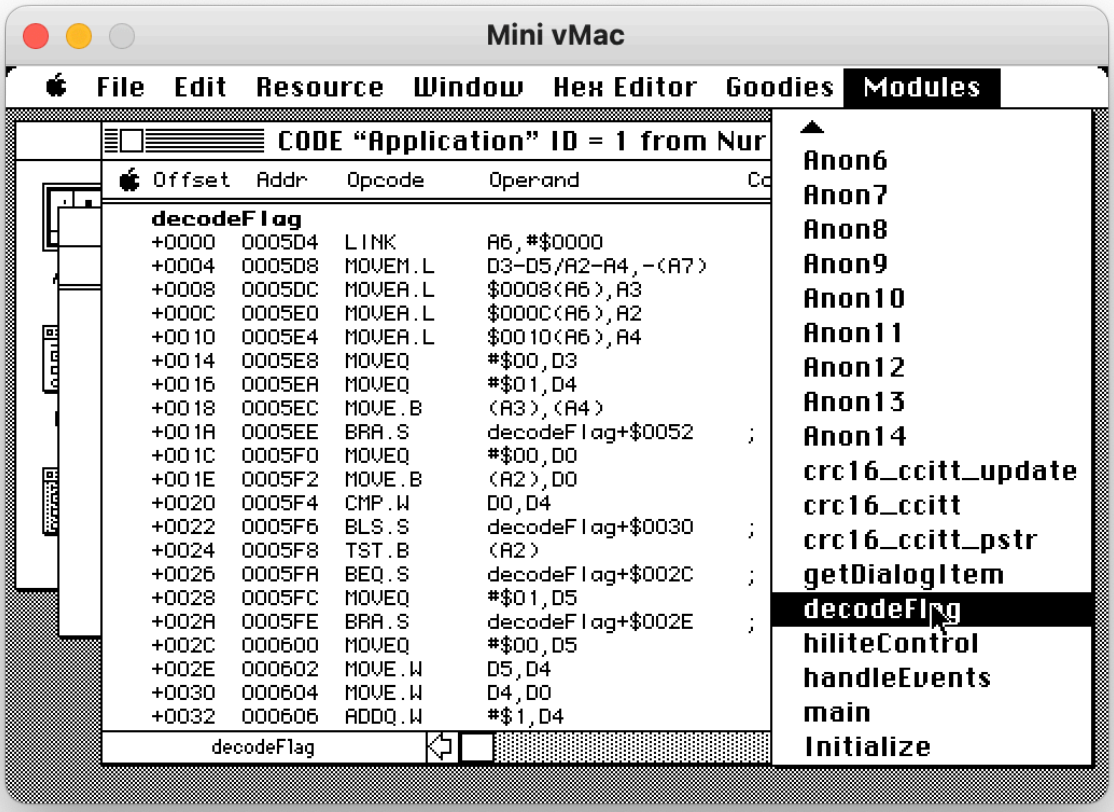
Here's your first hint: 1983 was a pretty good year in music.

Have fun, and enjoy the challenge! If you're still having trouble, maybe try asking the program if it has a bit of time for you; perhaps it will sing you a song.

- Dave Riley, July 2022There are actually several hints here, and the "first" one isn't really even the first, if you look closely. For the moment, let's take a look at the code with Super ResEdit's handy disassembler.



First, you can see that there are three code segments, and our application is by far the largest at... 2354 bytes. They really used to build 'em small, eh?

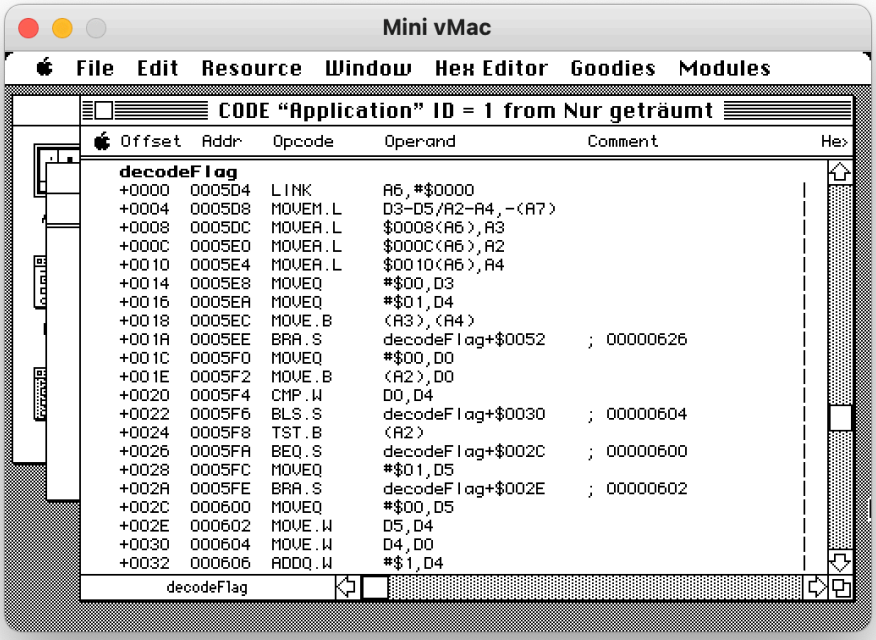


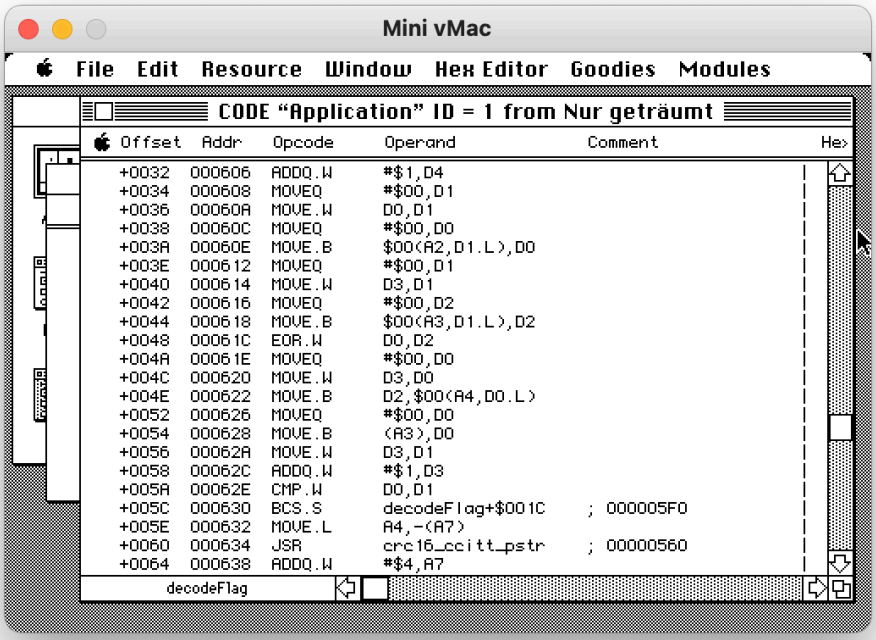
This build still has the debug symbols in (and the disassembler understands them), so many function names are still visible. We can see that there are some CRC functions as well as some UI handling functions. However, the decodeFlag function should be useful to look at.

Just as a note, there are other disassemblers for the Mac, but this is free. It's basic, but it's free. Steve Jasik's MacNosy is the absolute best Mac disassembler there is, providing a near IDA level of functionality while being much more conversant with Mac development idioms. Steve still sells it as a bundle with his excellent The Debugger and CoverTest (a code coverage package that's handy if you're doing classic Mac development) for $99, and it's absolutely worth the price (not least considering it used to cost $1500 back in the '90s).

Anyway, from here on out, you'll want a 68K assembler reference if you don't already know 68K assembly. It's a great language to learn! Personally, I like it a lot better than x86 assembly, though my favorite is still PowerPC.

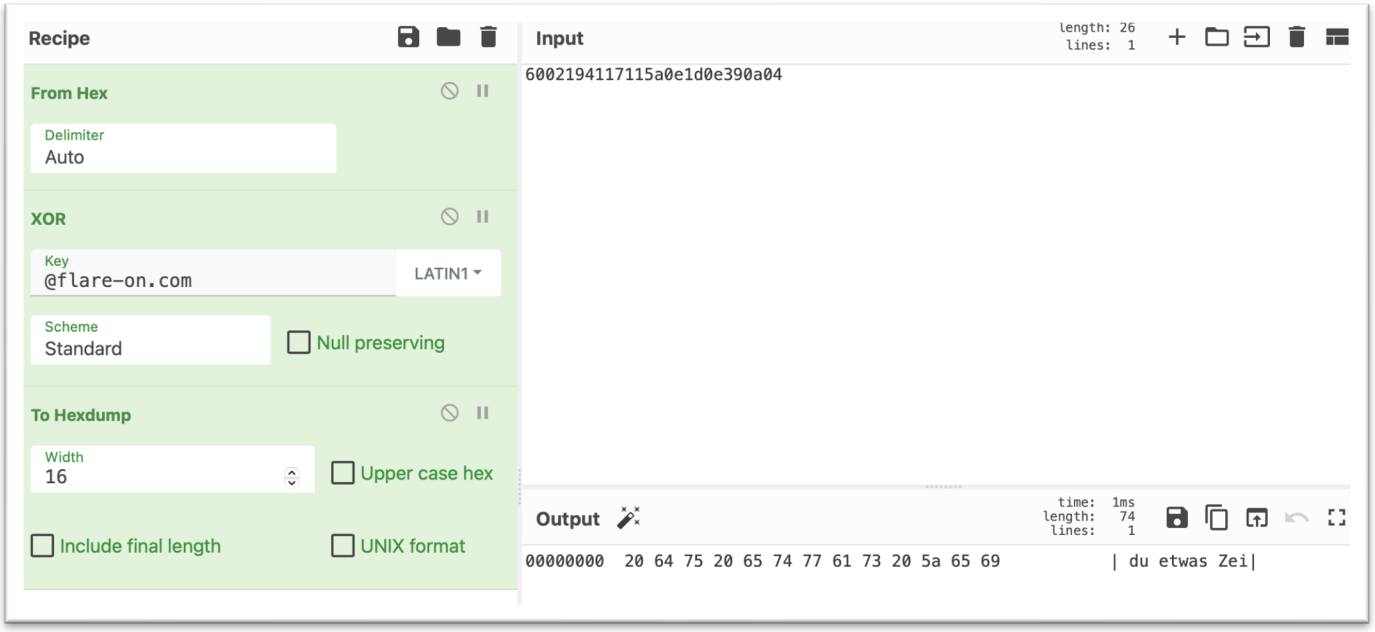
A quick primer: there are eight each of Data and Address registers (D0-D7 and A0-A7, all 32-bit), of which A7 is used as the stack register. For instructions of variable size, the prefixes .L (long), .W (word) and .B (byte) are suffixed to the instruction. Instructions with Q at the end are "quick", which means they use an immediate literal argument. Much like x86, there are two dialects of assembly; this disassembler has the target on the right, not the left (a la AT&T format). Immediate operands are prefixed with #, while the hexadecimal prefix is $.


Mini vMac
File Edit Resource Window Hex Editor Goodies Modules
CODE "Application" ID = 1 from Nur geträumt
Offset Addr Opcode Operand Comment He>
decodeFlag
+0000 0005D4 LINK A6,#$0000
+0004 0005D8 MOVEM.L D3-D5/A2-A4,-(A7)
+0008 0005DC MOVEA.L $0008(A6),A3
+000C 0005E0 MOVEA.L $000C(A6),A2
+0010 0005E4 MOVEA.L $0010(A6),A4
+0014 0005E8 MOVEQ #$00,D3
+0016 0005EA MOVEQ #$01,D4
+0018 0005EC MOVE.B (A3),(A4)
+001A 0005EE BRA.S decodeFlag+$0052 ; 00000626
+001C 0005F0 MOVEQ #$00,D0
+001E 0005F2 MOVE.B (A2),D0
+0020 0005F4 CMP.W D0,D4
+0022 0005F6 BLS.S decodeFlag+$0030 ; 00000604
+0024 0005F8 TST.B (A2)
+0026 0005FA BEQ.S decodeFlag+$002C ; 00000600
+0028 0005FC MOVEQ #$01,D5
+002A 0005FE BRA.S decodeFlag+$002E ; 00000602
+002C 000600 MOVEQ #$00,D5
+002E 000602 MOVE.W D5,D4
+0030 000604 MOVE.W D4,D0
+0032 000606 ADDQ.W #$1,D4
decodeFlag


Mini vMac
File Edit Resource Window Hex Editor Goodies Modules
CODE "Application" ID = 1 from Nur geträumt
Offset Addr Opcode Operand Comment He>
+0032 000606 ADDQ.W #$1,D4
+0034 000608 MOVEQ #$00,D1
+0036 00060A MOVE.W D0,D1
+0038 00060C MOVEQ #$00,D0
+003A 00060E MOVE.B $00(A2,D1.L),D0
+003E 000612 MOVEQ #$00,D1
+0040 000614 MOVE.W D3,D1
+0042 000616 MOVEQ #$00,D2
+0044 000618 MOVE.B $00(A3,D1.L),D2
+0048 00061C EOR.W D0,D2
+004A 00061E MOVEQ #$00,D0
+004C 000620 MOVE.W D3,D0
+004E 000622 MOVE.B D2,$00(A4,D0.L)
+0052 000626 MOVEQ #$00,D0
+0054 000628 MOVE.B (A3),D0
+0056 00062A MOVE.W D3,D1
+0058 00062C ADDQ.W #$1,D3
+005A 00062E CMP.W D0,D1
+005C 000630 BCS.S decodeFlag+$001C ; 000005F0
+005E 000632 MOVE.L A4,-(A7)
+0060 000634 JSR crc16_ccitt_pstr ; 00000560
+0064 000638 ADDQ.W #$4,A7
decodeFlag

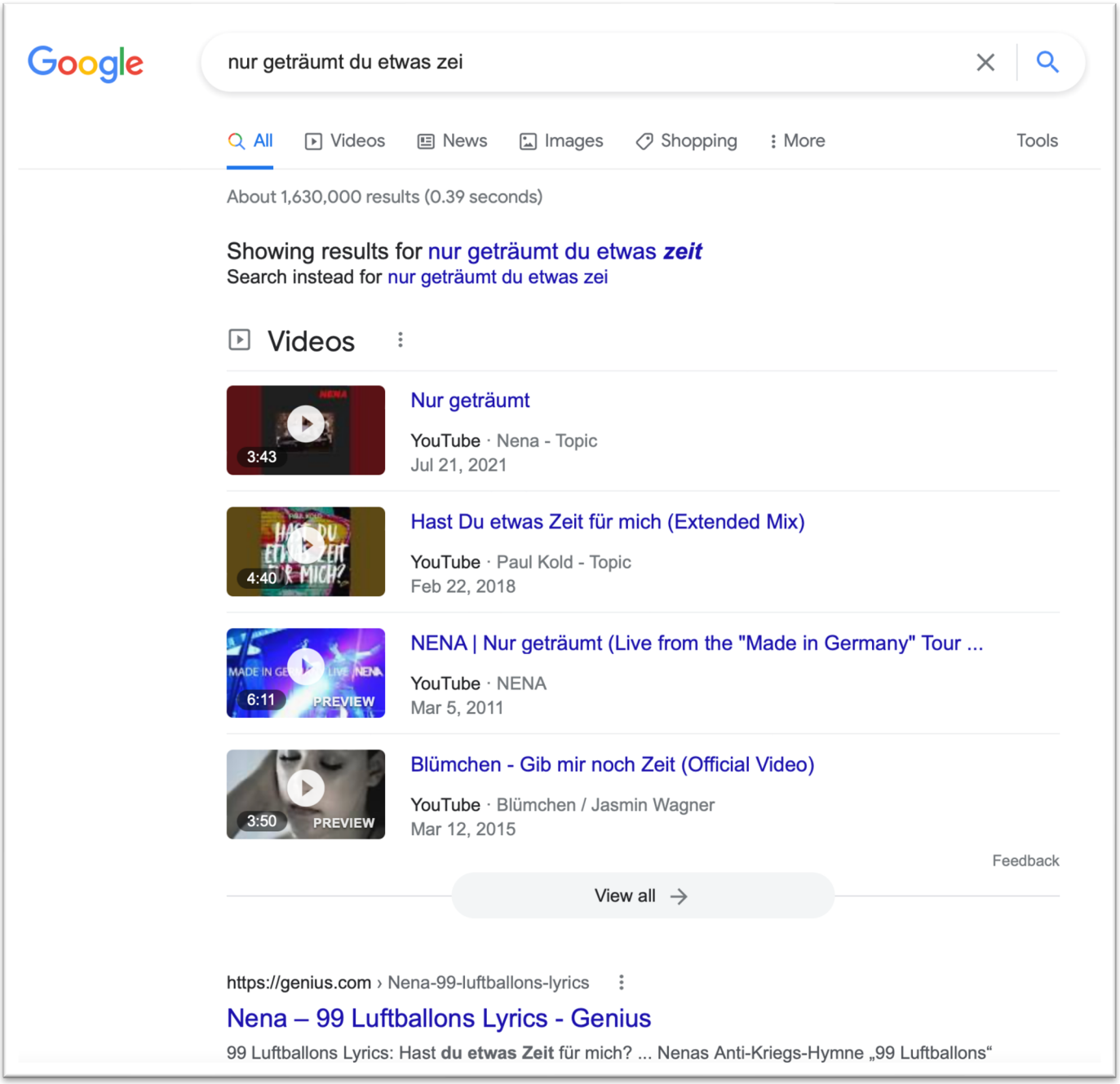
This is the meat of the decodeFlag function, where you can really tell that optimizations were not on at all; there's a lot of redundant code. The gist of it, though, is that the last argument is the destination string buffer, loaded into A4; the middle argument is the key string buffer, loaded into A2, and the first argument is the source string buffer. These are all Pascal strings, so they carry their lengths with them in the first byte. The function implements a simple keyed XOR encryption/decryption scheme (note that 68K's XOR is named EOR because the nomenclature had not fully settled by then). It then passes the decoded string to the CRC16-CCITT function for Pascal strings, and the return value of the CRC function (in D0) is returned from here. The last few lines (not in the above screenshots) are the cleanup epilogue.

Okay, so this is keyed XOR. That means it should be at least somewhat vulnerable to a known-plaintext attack. Fortunately, we know part of the plaintext: the end of the flag needs to end with @flare-on.com. We should be able to recover at least part of the key if we XOR the last bits of the encoded flag data with our known plaintext. ResEdit, unfortunately, doesn't provide any built-in tools for this, but you can cook up something quickly in CyberChef.



So, what we come out with is " du etwas Zei" (note the leading space), which is starting to look like a human language! In fact, it is a fragment of German, which is in keeping with the theme here.

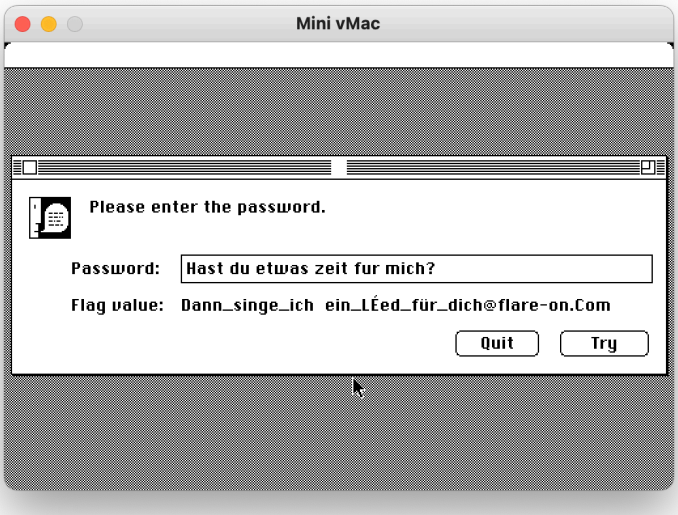
Where do we proceed here? Well, there are a few clues sprinkled about, including several exhortations to be curious. If you were to just paste the name of the program along with this fragment of key into Google, for example, you would find something quite revealing:



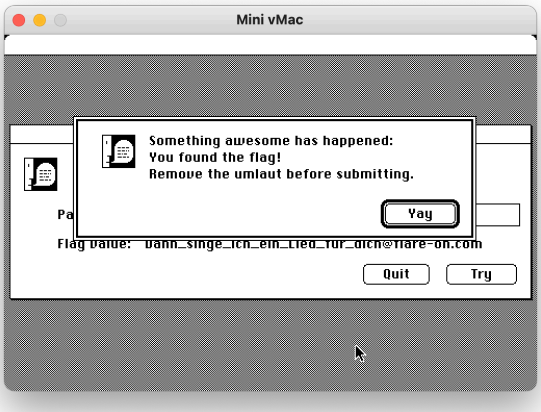
Check that bottom line. That looks kind of familiar, right? And the name of the flag resource was “99 Luftballons”, which indicates we may be on to something. Further, the text clue says the following:

If you're still having trouble, maybe try asking the program if it has a bit of time for you; perhaps it will sing you a song.

“Hast du etwas Zeit für mich?” translates to “Do you have some time for me?” Perhaps we should try that as a flag suggestion.



Hey, that's close! That's the second line of the song ("Then I'll sing a song for you"), but it's still not quite right, and the program hasn't let us know we found the real flag. Let's try it with proper capitalization and diacritic marks (on the Mac, an umlaut can be entered by typing Option-U and then the vowel you'd like, so Option-U and then U).



We got it! Like the alert box says, submit the flag without the umlaut (apologies for this; I painted myself into a corner with the solution before I got feedback that non-ASCII characters are very much not preferred for flags). Don't forget, the flag is case sensitive (as is German), so it is:

```
Dann_singe_ich_ein_Lied_fur_dich@flare-on.com
```

Closing Thoughts

This challenge was designed to have a few goals:

- Expose people to Classic Mac OS principles, including international character handling, as well as 68K assembly

- Use some OSINT research for both bringing up the Mac environment (kudos if you installed the system from floppy images, now imagine doing it at the speed of actual floppies) as well as researching the clues
- Provide a gentle encryption challenge containing a known-plaintext vulnerability, which is distressingly common in home-grown encryption schemes

I hope you enjoyed it!

Design Notes

I was a little uneasy requiring some pop culture knowledge to solve this, but I intentionally made it with pop culture with international reach (the German version of "99 Luftballons" was Nena's only #1 hit in the US, surpassing even the English translation, so it wasn't exactly obscure; "Nur geträumt" is another song from the same album, so I figured it should be easy enough to look up on search engines and find the opening line of "99 Luftballons").

I used keyed-XOR encryption for several reasons. First, as stated, it exposes the known plaintext vulnerability well, a common problem with simple encryption schemes; if you can determine what the plaintext at a given offset, you can very easily expose part of the key. Further, it rewards you for finding nearly right answers, which distinguishes it from something like RC4; with a typical encryption algorithm, one bit wrong in the key will result in garbage at the output, while with keyed XOR it will show you some slightly corrupted correct text. This is especially important for things that are case-sensitive, or sensitive to things like umlauts, especially if you can't copy and paste from the web (which most emulators won't do).

Originally, this challenge didn't have the CRC in it, but it became clear that it was much too easy to get a false-positive with the keyed XOR scheme, so the CRC was added to check for correct output. In theory, one could use the CRC to brute-force the key and/or the flag, but as stated above, given the length of both versus the 16-bit space of CRC16-CCITT, you'd be much more likely to wind up with a bunch of garbage than the actual flag, so it is robust against that. The core CRC update routine is written in hand-coded assembly because I didn't like the way it came out in C at all; look at the crc16_ccitt_update routine if you'd like to see some of the fun features of the 68K architecture.