Challenge 2 ("checksum")

Description

> We recently came across a silly executable that appears benign. It just asks us to do some math... From the strings found in the sample, we suspect there [is] more to the sample than what we are seeing. Please investigate and let us know what you find!

Writeup

This will be more of an anecdotal testimony of my naiveté than a straight-to-the-point writeup, as I'm sure there will be a lot of those out there.

We are given a zip with a standard PE Windows executable. The first step in my analysis is almost always to drop the binary into something like ExeinfoPE or Detect It Easy to detect any packers, crypters or specific compilers being used. Interestingly, in this instance, ExeinfoPE reported that the executable was compressed using something called "MEW 11 SE v1.1" — I spent some minutes looking for unpackers or just any code or blog posts explaining what this packer does, but to no avail.

As it turned out, the binary was not packed or tampered with in any meaningful way. The caveat was that it was not compiled from C or C++ code, but rather using the Go programming language and the Go compiler, which I have never dealt with before in my (albeit short) career as a reverse engineer. (As a matter of fact, I feel no shame in admitting I have never dealt with any Go code at all.)

The first challenge was hence to understand the Golang ABI, which differed in many aspects from the traditional Windows and Linux C/C++ ABIs I knew. I learned that parameters and return values are passed between caller and callee functions preferentially through registers — in fact quite many of them. Whereas for example the Windows 64bit C/C++ ABI makes use of 4 registers (rcx, rdx, r8 and r9, in that order) for passing arguments to functions and only one (rax) for return values, the Go ABI uses 9 (rax, rbx, rcx, rdi, rsi, r8, r9, r10 and r11, in that order) for both argument passing and return values. (Like in other ABIs, the remaining values are passed on the stack.) Furthermore, as Go supports returning multiple values from a function, it is often the case that a function will return either a meaningful value or `nil` as the first return value and an error code (where 0 means no error) as the second. All of this took some figuring out and getting used to, but once I had overcome this hurdle, reading compiled (non-stripped) Go code was actually surprisingly easy.

Analyzing the binary, I figured out (what I thought was) the basic high-level operation of the program:

1. A pseudorandom number n between 3 and 7 is chosen.
2. Then, n times, the user is asked to compute the sum of two numbers (pseudorandomly generated) and the answers are checked for correctness. If incorrect, the program terminates.
3. After n successful answers, the user is prompted for a "Checksum", which must be a 64-character ASCII hex string that is then parsed to a 32-byte array.

4. The encrypted flag stored in the binary is decrypted using the ChaCha20-Poly1305 authenticated encryption scheme (AEAD) with said byte array serving as the 256-bit key (and, simultaneously, the first 192 bits of the array serving as the nonce).
5. The SHA256 sum of the plaintext is computed and compared against the checksum provided by the user. The program is terminated in case of a mismatch. Note that such a check would make no sense in a real scenario, as the integrity of the plaintext is already guaranteed by the Poly1305 digest — if the decrypted data were wrong, the decryption procedure would report a failure straight away. Thus, this was nothing more than a note to the reverse engineer about what the checksum needed to be.

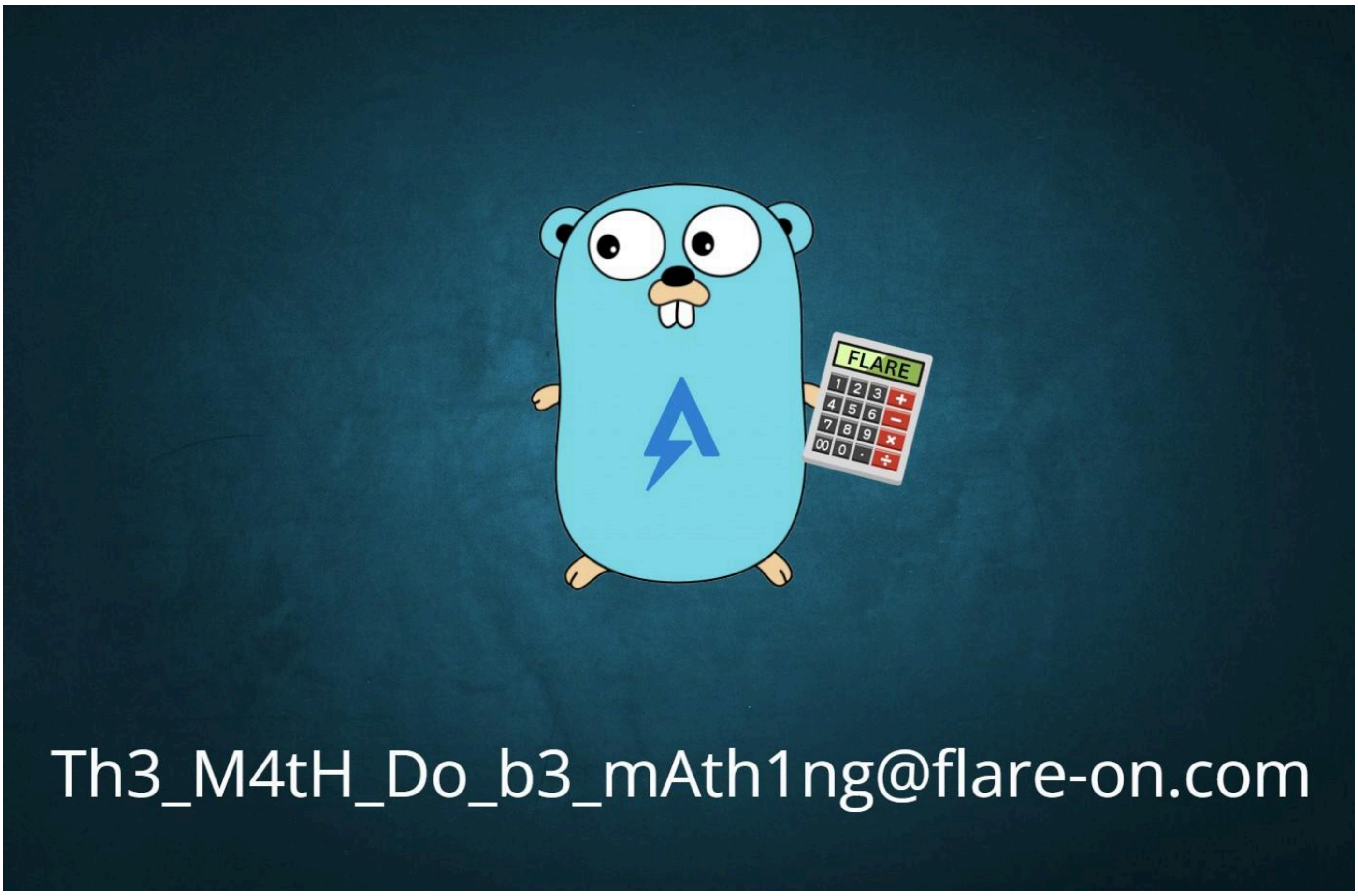
After learning this, I was baffled. So baffled that I didn't realize that I had skipped the analysis of one more function, called `main_a`, which was being called at the very end before the program would report success. I even went as far as to look into the ChaCha20 key scheduling algorithm to look for a potential way to reduce the keyspace using the combined fact that the nonce was equal to the key and that I knew the first 2 bytes of the plaintext (since the program clearly expected the plaintext to be a JPEG file, the first two bytes would be `FF D8`). Of course, I soon realized that being able to reduce the keyspace in any meaningful way would be a massive flaw in the cipher, and it was just as clear to me that this was way too advanced for the second challenge. I finally remembered I had skipped the `main_a` function and came back to it.

The workings of `main_a` were stupidly simple. The original buffer with the input (still in ASCII) was "XOR-encrypted" with the key `FlareOn2024`, then base64-encoded and compared to the fixed string `cQoFRQErX1YAVw1zVQdFUSxfAQNRBXUNAxBSe15QCVRVJ1pQEwd/WFBUAlElCFBFUnlaB1ULByRdBEFdfVtWVA==` located in the `.rdata` section. If the two strings compared equal, the program printed a success message and dropped a JPEG file containing the flag.

So after all these cryptographic shenanigans, the solution was handed to us on a silver platter. All we had to do is reverse these last steps, which was trivial:

```
from base64 import b64decodect =
b64decode('cQoFRQErX1YAVw1zVQdFUSxfAQNRBXUNAxBSe15QCVRVJ1pQEwd/WFBUAlElCFBFUnlaB1ULByRdBEFdfVt
WVA==')key = ('FlareOn2024' * 10)[:64].encode()pt = "".join([chr(ct[i] ^ key[i]) for i in
range(len(ct))])print(pt)
```

Which prints `'7fd7dd1d0e959f74c133c13abb740b9faa61ab06bd0ecd177645e93b1e3825dd'` — the desired checksum. After entering this checksum into the program, the flag is dropped into the `AppData/Local` user directory.



A humbling lesson

While this crackme took me longer to solve than I would have liked, it was well worth it, as I learned valuable things along the way. I now know how to approach reversing binaries compiled using Go, I understand the ABI, calling conventions and even the implementation of some types (like slices). I know where to find the Go standard library source code and that `HChaCha20` and `hChaCha20` are two completely different functions with different parameters (I guess Google developers are just built different).

But most importantly, I was reminded that while I was able to reverse engineer the program — especially the more complex parts that require non-trivial cryptographic knowledge — with relative ease, I still have much to learn in terms of the approach to take when reversing. If I had started working back from the end goal instead of from the start, I likely would have found the solution a lot sooner.

Revision #2
Created 23 December 2024 20:55:14 by Annatar
Updated 24 December 2024 16:28:07 by Annatar