Flare-On 3: Challenge 7 Solution - hashes.exe

Challenge Author: Alex Berry

This challenge starts with a single executable file: hashes. Opening this file in a hex editor quickly reveals that it is a 32-bit ELF for Linux binary. Loading the binary on a Linux system and running ldd on the binary reveals that the binary is linked with `libgo.so.7`, as shown in Figure 1.

```
# ldd hashes
        linux-gate.so.1 =>  (0xb77eb000)
        libgo.so.7 => /usr/lib/i386-linux-gnu/libgo.so.7 (0xb6c4b000)
        libgcc_s.so.1 => /lib/i386-linux-gnu/libgcc_s.so.1 (0xb6c2d000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb6a71000)
        /lib/ld-linux.so.2 (0x80061000)
        libpthread.so.0 => /lib/i386-linux-gnu/libpthread.so.0 (0xb6a54000)
        libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0xb6a07000)
```

Figure 1: ldd output

This means we have a 32-bit ELF binary that was written in the go programming language. If we run the program, it quickly crashes reporting an error to the console and if it is run with one or more arguments it prints a message; these outputs are shown in Figure 2.

```
#./hashes
panic: runtime error: index out of range

goroutine 16 [running]:

goroutine 18 [finalizer wait]:
created by runtime_createfing
        ../../../src/libgo/runtime/mgc0.c:2572
#./hashes test
Work on your Hash F00!
```

Figure 2: Inital Program Output

Loading the binary in IDA Pro, IDA correctly recognizes the main function for the binary as `main_main`. Stepping through this function we come upon a comparison between 1 and `dword_805022C`. We can quickly recognize that `dword_805022C` is the argument count, as a go runtime error is produced when this value is less than or equal to 1 and the program continues to run otherwise. The go arguments are stored as an array of pointers to string structures in memory. The string structure contains a pointer to the string bytes followed by the length of the string. The sequence of instructions shown in Figure 3 loads the string length into `var_28`.

```
.text:0804A06B lea     eax, [ebx+esi]
.text:0804A06E mov     eax, [eax+4]
.text:0804A071 mov     [ebp+var_28], eax
```

Figure 3: Getting the argument length

A quick check of the cross-references to the string length leads us to a comparison between `var_28` and `0x1F`. The length of the argument does not equal `0x1E` then hashes prints the "`Work on your Hash F00!`" message. We now know the first argument must have a length of 0x1E (30). If the length check is passed, the string is passed into sub_8049F6D. This subroutine verifies that all the characters of the input string are in the character set "`abcdefghijklmnopqrstuvwxyz@-._1234`".

The binary then slices the input string into five substrings of length six and passes them one at a time into `sub_8049CFB`. After converting the string to a byte array this function performs a SHA1 hash of the byte array, a SHA1 on the resulting digest, and a SHA1 on the resulting digest again. This operation is shown in the python code shown in Figure 4. The resulting digest is returned by the function.

```
def hash_func(somestr):
      for i in range(3):
            somestr = SHA.new(somestr).digest()
      return somestr
```

Figure 4: String slice hashing function

After each slice is hashed, the resulting digests are concatenated together to produce a 100-byte array. The pointer to the final byte array is stored on the stack at `var_C4` with the length stored at `var_C8`.

During the initialization we saw the binary create a go channel at `0804A0DB` with the resulting reference stored at `var_2C`. Go channels are used as pipes to communicate between go routines. Later at `0804A296` we see the initialization of the go routine. The structure containing a reference to the go channel, the output byte array length and the first byte of the array, shown in Figure 5, are passed into the initialization routine along with a pointer to `sub_804A53B`.

```
.text:0804A27F mov     [edx], eax       ; go_routine var_2C ref
.text:0804A281 mov     [edx+4], esi     ; first element of hash array
.text:0804A284 mov     [edx+8], ebx     ; hash array length
```

Figure 5: go routine arguement structure load

Examining sub_804A53B we can see that when it is called it is provided a similar structure that is unpacked and provided to sub_8049EE7 as shown in Figure 6.

```
.text:0804A554 mov     eax, [ebp+arg_4]
.text:0804A557 mov     ecx, [eax+8]     ; hash array length
.text:0804A55A mov     eax, [ebp+arg_4]
.text:0804A55D mov     edx, [eax+4]     ; first element of byte array
.text:0804A560 mov     eax, [ebp+arg_4]
.text:0804A563 mov     eax, [eax]       ; var_2C channel reference
.text:0804A565 sub     esp, 0Ch
.text:0804A568 push    ecx
.text:0804A569 push    1CDh
.text:0804A56E push    1000h
.text:0804A573 push    edx
.text:0804A574 push    eax
.text:0804A575 call    sub_8049EE7
```

Figure 6: go channel function call

From this point forward ___go_receive calls sub_804A53B and sub_8049EE7 then calls ___go_send_small to return the resulting value. The function at sub_8049EE7 is a pseudo random number generator that can be represented by the python generator shown in Figure 7.

```
def prng(seed, max=0x1000, step=0x1CD, tlen=100):
      start = seed
      for i in range(tlen):
            start = (start + step) % max
            yield start
```

Figure 7: Python generator form of go routine channel

The resulting value of the channel is then used as an index into 4096-byte array at byte_804BB80. We now know that the 100-byte hash digest sequence of the input is stored in 4096-byte array and we can use the generator code above to narrow down the number of 100-byte sequences that we need to examine. Specifically, we know that the following must be true byte_804BB80[cyclic_gen(seed)[0]] == seed, as the seed is the first element of the concatenated hashes. The IDA Python code shown in Figure 8 performs this search and prints the resulting 100-byte arrays.

```
def search():
      for i in range(256):
            next_idx = prng(i)
            d = "" + chr(i)
            if Byte(0x0804BB80 + next_idx.next()) == i:
                  for z in next_idx:
                        d += chr(Byte(0x0804BB80 + z))
                  print d.encode("hex")

------ Result ------
3cab2465e955b78e1dc84ab2aad1773641ef6c294a1bf8bd1e91f3593a6ccc9cc9b2d5682e62244f9e6
061a36250e1c47e69f0312db4e561528a1fb506046b721e18e20b841f497e257753b2314b866ccc7208
42d0884da08e26d9fccb24bc9c27bd254e
7afc01ff7c2ae6768ad7281b1025c7d64e9a905fef16ec2a43f5d840efdae1aaaabd7dc3b7670810a4f
6f80389125aad77e918db77a466e5ab7db10ffe140ae073bca6e0071d7d1c29a5fa1b73a99a06471450
5cf92f2fbaeaac1059a5613a3928285b88
f91727f8892e34f2be1786fa115bc4ad621dd4ac92e4de8810744a70338e854adc7803e1eab70941387
72f47a05e778af70a1f1d5c8674b6fa63f4127cb25b5598ea410086a995d0c41770b46414599bee613d
1a1a64e064c31b9222f70566b9d6939c52
```

Figure 8: Python code to search for hash array sequences and resulting values

To further narrow the search, we can use the knowledge that all keys end with `@flare-on.com` to produce the final two hashes shown in Figure 9.

```
print hash_func("flare-").encode("hex")
print hash_func("on.com").encode("hex")

------ Result ------
06046b721e18e20b841f497e257753b2314b866c
cc720842d0884da08e26d9fccb24bc9c27bd254e
```

Figure 9: Hashing flare-on.com

These two hashes are the trailing data for the first sequence found in Figure 8, so we know that was the correct sequence. We can break the sequence into five twenty byte SHA1 outputs shown in below, with the known inputs and the unknown hashes that still need to be solved for.

```
3cab2465e955b78e1dc84ab2aad1773641ef6c29
4a1bf8bd1e91f3593a6ccc9cc9b2d5682e62244f
9e6061a36250e1c47e69f0312db4e561528a1fb5
06046b721e18e20b841f497e257753b2314b866c  # hash_func("flare-")
cc720842d0884da08e26d9fccb24bc9c27bd254e  # hash_func("on.com")
```

Figure 10: SHA1 hashes from the correct byte array sequence

Now we can implement a brute forcing algorithm, shown in Figure 11, to search for the final strings that result in the first three hash values.

```
import itertools

goal = [
  "3cab2465e955b78e1dc84ab2aad1773641ef6c29".decode("hex"),
  "4a1bf8bd1e91f3593a6ccc9cc9b2d5682e62244f".decode("hex"),
  "9e6061a36250e1c47e69f0312db4e561528a1fb5".decode("hex"),
]

alpha = "abcdefghijklmnopqrstuvwxyz@-._1234"

cur = 0
for tup in itertools.product(alpha, repeat=6):
      if not goal:
            print "Done"
            break;
      chash = hash_func("".join(tup))
      if chash in goal:
            print "".join(tup), chash.encode("hex"),
            goal.remove(chash)
            print len(goal)

------ Result ------
h4sh3d 3cab2465e955b78e1dc84ab2aad1773641ef6c29 2
4sh3s@ 9e6061a36250e1c47e69f0312db4e561528a1fb5 1
_th3_h 4a1bf8bd1e91f3593a6ccc9cc9b2d5682e62244f 0
```
Figure 11: Brute Forcing Algorithm

Bringing it all together, we can now pass the resulting string to the original program producing the output shown below.

```
#./hashes h4sh3d_th3_h4sh3s@flare-on.com
You have hashed the hashes! h4sh3d_th3_h4sh3s@flare-on.com
```
Figure 12: Answer Key