Lab 2 Report

by Eli K. Martin for CS 4173

Table of Contents

Lab source: [MD5 Collision Attack Lab](#)

Links: [Table of Contents](#) | [References](#)

Pre-Lab Setup

For Lab 2, we will be using the same virtual machine as Lab 1 (the SEED pre-built Ubuntu 20.04 VM). I verified it was still installed and functioning correctly, and I verified that the “md5collgen” program that we will be using was installed on the VM as well.

Task 1: Generating Two Different Files with the Same MD5 Hash

For this task, we use the included md5collgen program to create two different files with the same MD5 hash. I started by creating a prefix text file “prefix_a.txt” with some arbitrary text (Fig 1.1). I used this with md5collgen to create “out1a.bin” and “out2a.bin” (Fig 1.2) which I then viewed in the bless editor. They were different in a few places, which was confirmed by the diff command. Finally, I ran the md5sum command to find the MD5 hashes of both output files, and saw that they were identical (Fig 1.3).

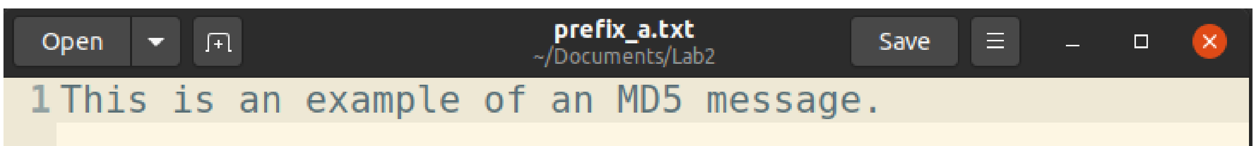

Fig 1.1 The first prefix used to create identical hashes.

```
[10/08/22]seed@VM:~/.../Lab2$ md5collgen -p prefix_a.txt -o out1a.
bin out2a.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1a.bin' and 'out2a.bin'
Using prefixfile: 'prefix_a.txt'
Using initial value: 3de612cbedddb9f010e0b5364047c0a8

Generating first block: ....
Generating second block: W.
Running time: 19.0436 s
[10/08/22]seed@VM:~/.../Lab2$ bless
```

Fig 1.2 The mdcollgen command used to create the first two files that should have identical hashes.

Links: [Table of Contents](#) | [References](#)

```
[10/08/22]seed@VM:~/.../Lab2$ md5sum out1a.bin out2a.bin
7b46bdcbf6a2d34c0232deb01b9c5f6a  out1a.bin
7b46bdcbf6a2d34c0232deb01b9c5f6a  out2a.bin
```

Fig 1.3 The two files created by md5collgen have the same MD5 hashes.
This shows that given some prefix, we are able to find two messages including that prefix that hash to the same message. This violates the weak collision property, which states that for a given message, it is difficult to find a second one with an identical hash. If we choose our message to be one output file from md5collgen, we know that the second output file will have an identical hash.
Now we will look at how those two output files differ. I copied the hexadecimal representation from the bless editor of each file and put it in a text comparison editor (Fig 1.4 and Ref 1).

Removed Added

```
1  54 68 69 73 20 69 73 20 61 6E 20 65 78 61 6D 70 6C 65 20 6F 66 20 61 6E 20 4D 44 35 20 6D 65 73 73 61 67 65 2E
   0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A6 5F F2 1D DE 84 B1 33 F5 BE
   BB 45 3A B4 2A DC 23 FB 4A B3A F2 D3 76 99 1A F2 80 A5 D7 20 8C 45 AA 18 C3 E8 E2 22 74 8B 0F 28 6C C2 CA
   6E7 5A 2E AB E0 32 7B E7 19 28 56 81 48 3D 808 42 53 A6 CC E6 8C 02 E4 A9 D7 13 1E 23 7D F2 C9 22 74 F9 EB E0
   53 29 4C7 10 47 3D 4E 04 6D 20 98 CF ED BD AF B9 5F 3E D7 5A 56 71 A7 3E 36 01 2A 9B B38 FA 7C DF 00 24 D7
   6E B9 C3 F7 97 56 DB 6EF 55 45 EE ED bin12
```

Fig 1.4 The text comparison editor shows the difference in output1a.bin (red) and output 2a.bin (green).

We can see that the following bytes were edited between output1a and output2a: 0x53 (B to 3), 0x6d (6 to E), 0x7b (8 to 0), 0x93 (4 to C), 0xad (B to 3), and 0xbb (6 to E). These differences come from a 38-byte original file (prefix_a.txt). If we run the command to generate the output binaries again, we can see we get two different files, out3a.bin and out4a.bin, that have identical MD5 hashes as well, although they are different than the hashes for out1a and out2a (Fig 1.5). Comparing out4a and out1a, we can see that the appended characters are quite different, although our original message is the same (Fig 1.6). We can also see that out3a and out4a have 7 bytes different (Fig 1.7), while out1a and out2a had only 6 bytes different.

```
[10/08/22]seed@VM:~/.../Lab2$ md5collgen -p prefix_a.txt -o out3a.
bin out4a.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out3a.bin' and 'out4a.bin'
Using prefixfile: 'prefix_a.txt'
Using initial value: 3de612cbedddb9f010e0b5364047c0a8

Generating first block: ......
Generating second block: S11..................................
Running time: 33.4727 s
[10/08/22]seed@VM:~/.../Lab2$ md5sum out3a.bin out4a.bin
1b85f8a79b628bc4fd37b91caf637aa9  out3a.bin
1b85f8a79b628bc4fd37b91caf637aa9  out4a.bin
```

Fig 1.5 The hashes shown here for out3a and out4a are identical to each other, but different than the ones shown in Fig 1.3 for out1a and out2a.

Removed Added

```
1  54 68 69 73 20 69 73 20 61 6E 20 65 78 61 6D 70 6C 65 20 6F 66 20 61 6E 20 4D 44 35 20 6D 65 73 73 61 67 65 2E
   0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A6 5F F2 1D DE 84 B18F D6 55
   33 F95 BE BB 45 3A B4 2A DC 2EB FB B3 F6 89 D0 84 D3 FBF 4A BA F2 D3 76 99 1A F2 80 A5 D7 20 8C 45 AA 18 C3
   E8 E2 22 74 8B 0F 28 6C C2 CA 67 5A 2E AB E0 32 7B E7 11 F5 3F 73 6E 0F 05 80 91 13 2E EE 42 3E A9 21 AE AE CE
   E9 E3 BF FE 66 3E 84 67 60 F2 73 CC 3F 0B 8B C2 CE D3 39 2E8 56 81 48 3D 88 42 53 A6 CC E6 8C 02 E4 A9 D7 13
   1E 23 7D F2 C9 22 74 F9 EB E0 53 2969 54 6C 83 9E EB FA B4 7C 68 BA 21 13 50 0A DE C1 3F 67 84 AE F9 FE 47B
   10 47 3D 4E 04 6D 20 98 CF ED BD AF B9 5F 3E D7 5A 56 71 A7 3E 36 01 2A 9B B8 FA 7C DF 00 24 D7 6E B9 C3 F7
   97 56 DB 6F 55 45 EE EDD 8B B0 CD A1 59 BF 26 AB 8C 3F 86 17 C0 5F 5F 8C 4D C1 E2 2C C1 0E 84 28 83 96 57 B1
   E7 90 F0 4E 70 65 5C F8 6C 77 8C CE 20 51 6E BD 23 72 E8 42 75 F1 bin14
```

Fig 1.6 A comparison of out1a.bin (green) and out4a.bin (red). We can see the appended text is almost completely different; there are only a few characters in grey indicating they are the same between files. However, we see that our original prefix is still the same across both iterations of md5collgen.

```
1  54 68 69 73 20 69 73 20 61 6E 20 65 78 61 6D 70 6C 65 20 6F 66 20 61 6E 20 4D 44 35 20 6D 65 73 73 61 67 65 2E
   0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 8F D6 55 33 95 EB FB B3 F6 89
   D0 84 D3 FF 41 F5 3F 73 6E 80F 05 80 91 13 2E EE 42 3E A9 21 AE AE CE E9 E3 BF FE 66 3E 84 67 60 F2 73 CC B3F
   0AB 8B C2 CE D3 39 E8 69 54 6C 83 9E EB 7FA B4 7C 68 BA 21 13 50 0A DE C1 3F 67 84 AE F9 FE 4B 1D 8B B0 CD
   A1 59 3BF 26 AB 8C 3F 86 17 C0 5F 5F 8C 4D C1 E2 2C C1 0E 84 28 83 96 57 B1 E7 90 F0 C4E 70 65 5C F8 6C 77 8C
   CE 20 51 6E BD 23 F72 E8 42 75 F1 bin34
```

Fig 1.7 This shows the difference between out3a.bin (red) and out4a.bin (green).

Links: Table of Contents | References

We also see that the padding of 0's (which is 26 bytes long) appended to our messages by md5collgen is the same. Looking in our files' properties, we can see that all of the output files are 192 bytes. Our prefix file was 38 bytes. This leaves roughly 128 bytes of information that is different between the output files. It seems that the 0s padding was to bring the total length of the input up to 64. We will next test that by creating an input file that is exactly 64 bytes (which is 64 characters) long, called "prefix_b.txt" (Fig 1.8). Then I created two sets of files with md5collgen like we did for the first input (Fig 1.9). Finally, I found the MD5 hash of these created files (Fig 1.10).

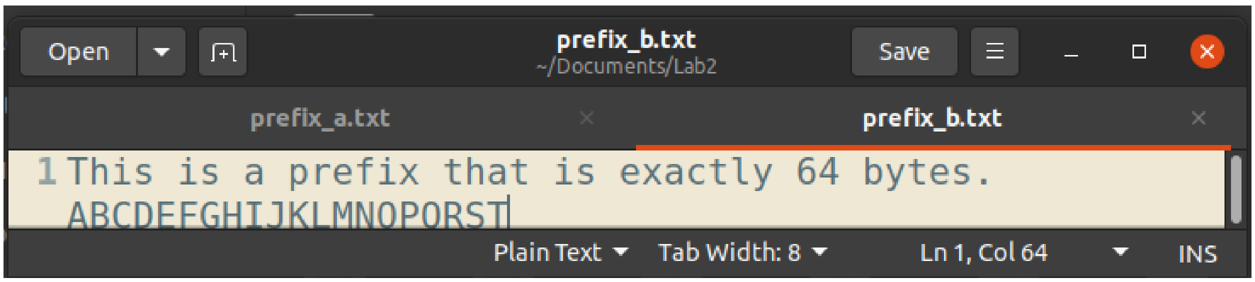

Fig 1.8 The text file "prefix_b.txt" which is exactly 64 bytes.

```
[10/08/22]seed@VM:~/.../Lab2$ md5collgen -p prefix_b.txt -o out1b.b
in out2b.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1b.bin' and 'out2b.bin'
Using prefixfile: 'prefix_b.txt'
Using initial value: 85fa4314077a78f3516cc722f8ea8e7c

Generating first block: ....
Generating second block: S11.......................................
.........................
Running time: 18.9699 s
[10/08/22]seed@VM:~/.../Lab2$ md5collgen -p prefix_b.txt -o out3b.b
in out4b.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out3b.bin' and 'out4b.bin'
Using prefixfile: 'prefix_b.txt'
Using initial value: 85fa4314077a78f3516cc722f8ea8e7c

Generating first block: ............
Generating second block: W...
Running time: 6.2423 s
[10/08/22]seed@VM:~/.../Lab2$
```

Links: [Table of Contents](#) | [References](#)

Fig 1.9 Creating output files for the prefix_b.txt file.

```
[10/08/22]seed@VM:~/.../Lab2$ md5sum out1b.bin out2b.bin
5457cf57d768750cf58453140cf3178a  out1b.bin
5457cf57d768750cf58453140cf3178a  out2b.bin
[10/08/22]seed@VM:~/.../Lab2$ md5sum out3b.bin out4b.bin
862dced9046f9235a0a9ab8384ffd907  out3b.bin
862dced9046f9235a0a9ab8384ffd907  out4b.bin
[10/08/22]seed@VM:~/.../Lab2$
```

Fig 1.10 The hashes for all the output files created from the prefix_b.txt file.

As we can see, there is a similar effect as with the first prefix text. Both out1b and out2b have the same hash, and out3 and out4b have the same hash, but the two pairs have very different hashes. We can see in the output files that this is reflected in the text (Fig 1.11 and Fig 1.12).

```
1  54 68 69 73 20 69 73 20 61 20 70 72 65 66 69 78 20 74 68 61 74 20 69 73 20 65 78 61 63 74 6C 79 20 36 34 20 62
   79 74 65 73 2E 20 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 0A 55 F8 A4 9A AE 46 4C 25 77 0B
   06 53 B2 76 E4 C4 63 D3 E3 E6A E4 E0 73 54 08 8E 80 B2 09 BA B3 B2 62 B5 0B C4 72 AE 41 CE EF 5D EB 5C F0 081
   EC 53 FB 70 E8 9F 1A 6F DD 2D 9A A4 A9 A2F 24 D0 54 97 2C AF 92 BC EF D6 9A 80 98 45 E9 6D C0 83 BC 8E BD 1F
   5E 7F4 27 A8 09 1E 74 47 80 4B 4F 5D 4D C1 E1 5C B9 0A 82 6F 92 9F A8 BD EF 98 34 80F 6B 4B 12 FB B0 EF C3 C1
   0E 99 CD 28 72 D5F 8A 08 D4 31 out12b
```

Fig 1.11 The comparison of out1b.bin (red) and out2b.bin (green) where we can see there are still only a few bytes different between them, similar to out1a.bin and out2a.bin.

```
54 68 69 73 20 69 73 20 61 20 70 72 65 66 69 78 20 74 68 61 74 20 69 73 20 65 78 61 63 74 6C 79 20 36 34 20 62
79 74 65 73 2E 20 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 0A DD A0 99 1E 8F 4F 6B 9C A2 77
19 89 72 05 BD F8 4D 0D 06 4C8 9C 40 9C F9 2D 5A 41 BD 68 84 BC 40 67 9C 04 EF CC 0A E4 6E 8B 04 4D 64 88 6ED
69 63 CA C8 75 CA D7 29 36 3F F0 FE 4A 6EC A8 27 9A 8A F5 7A F0 6B E5 EC 31 67 79 4E 2E C1 4B A7 65 E2 7F 34
31 C44 73 37 8B 0B EC BC A3 E6 D1 FA AD 0E 85 FF FE 02 64 B6 31 87 7E 6B 01 2A A3 7F0 5ED 5E F0 D4 5D C9 64
C5 63 75 40 DC 1F 7F1 8A FA 04 18 out34b
```

Fig 1.12 The comparison of out3b.bin (red) and out4b.bin (green) is again similar to previous comparisons.

The big difference between the two prefix's output files is that, as expected, when the input is exactly a multiple of 64 bytes, the output does not include any 0s for padding. With these observations, we can now clearly answer all 3 questions asked in the lab documentation:

Links: Table of Contents | References

Question 1

If the length of your prefix file is not multiple of 64, what is going to happen?
If our prefix file is not a multiple of 64, the mdcollgen program will pad it with 0s so that it is before appending the other bytes to the file.

Question 2

Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.
We saw that this creates a very similar file to a prefix file with under 64 bytes, but without any padding.

Question 3

Are the data (128 bytes) generated by md5collgen completely different for the two output files? Please identify all the bytes that are different.
There are approximately 6-7 bytes different each time between the two generated files, which were highlighted in the previous figures.

Task 2: Understanding MD5's Property

In this task, we are supposed to design an experiment to demonstrates the following property of MD5: If we have two messages M and N such that MD5(M) = MD5(N), and then we concatenate those message with T to get (M||T) and (N||T), then MD5(M||T) = MD5(N||T). I will demonstrate this using messages generated the same way as in the last task.
First, I will choose a prefix text file. Next, I will use md5collgen to generate two messages starting with this prefix that are guaranteed to have the same hash (as demonstrated in the last task). These will represent M and N. Then, I will choose a binary file to concatenate with both M and N to represent T. Finally, I will use md5sum to find the MD5 hash of both (M||T) and (N||T) and verify that these hashes are the same and thus that the property is upheld.
I started by choosing the prefix file which I called prefx_c.txt (Fig 2.1).

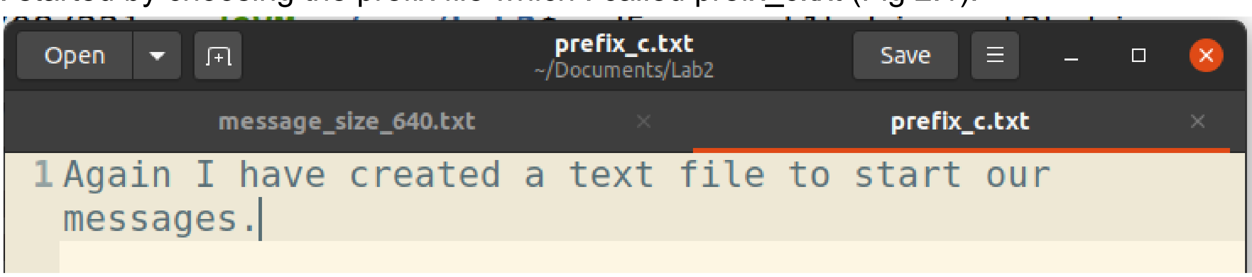

Fig 2.1 This is the file prefix_c.txt which will be our messages' prefix.

Links: [Table of Contents](#) | [References](#)

Next, I used md5collgen to generate the two unique messages out1c.bin and out2c.bin to serve as our M and N (Fig 2.2).

```
[10/08/22]seed@VM:~/.../Lab2$ md5collgen -p prefix_c.txt -o out1c.b
in out2c.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1c.bin' and 'out2c.bin'
Using prefixfile: 'prefix_c.txt'
Using initial value: c509b9037e2901428d12780870a64906

Generating first block: ...................................
...................................................................
.........................................
Generating second block: W.........................................
........
Running time: 121.911 s
[10/08/22]seed@VM:~/.../Lab2$ diff out1c.bin out2c.bin
Binary files out1c.bin and out2c.bin differ
[10/08/22]seed@VM:~/.../Lab2$ 
```

Fig 2.2 Creating the outputs from the prefix and verifying that they differ.

Then I created a binary file to concatenate with the other files, called “concatenation.bin” (Fig 2.3). I used the “cat” command to add it to the end of both out1c.bin and out2c.bin to create concat1.bin and concat2.bin respectively (Fig 2.4, Fig 2.5, and Fig 2.6).

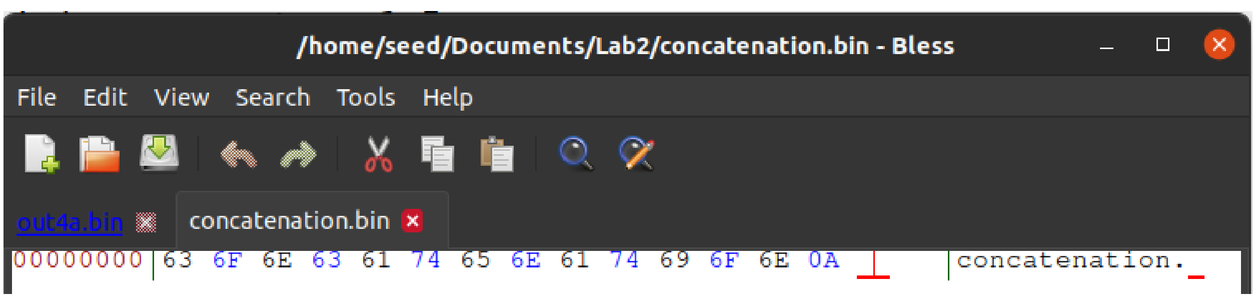

Fig 2.3 The binary file “concatenation.bin” I created to use for concatenation.

Links: [Table of Contents](#) | [References](#)

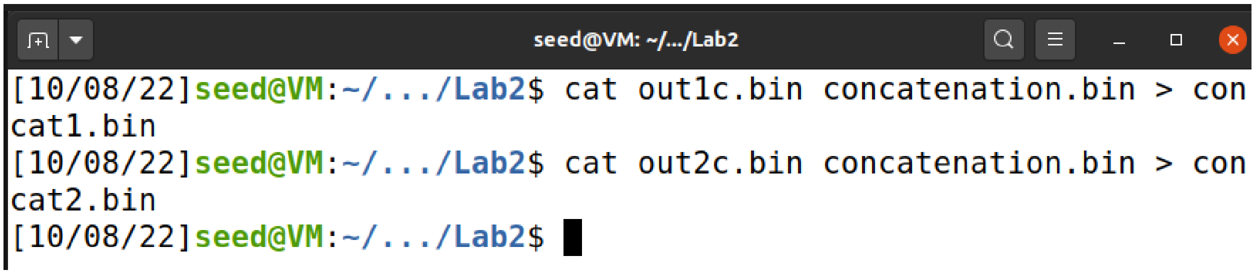

Fig 2.4 Concatenating each output file with the concatenation file.

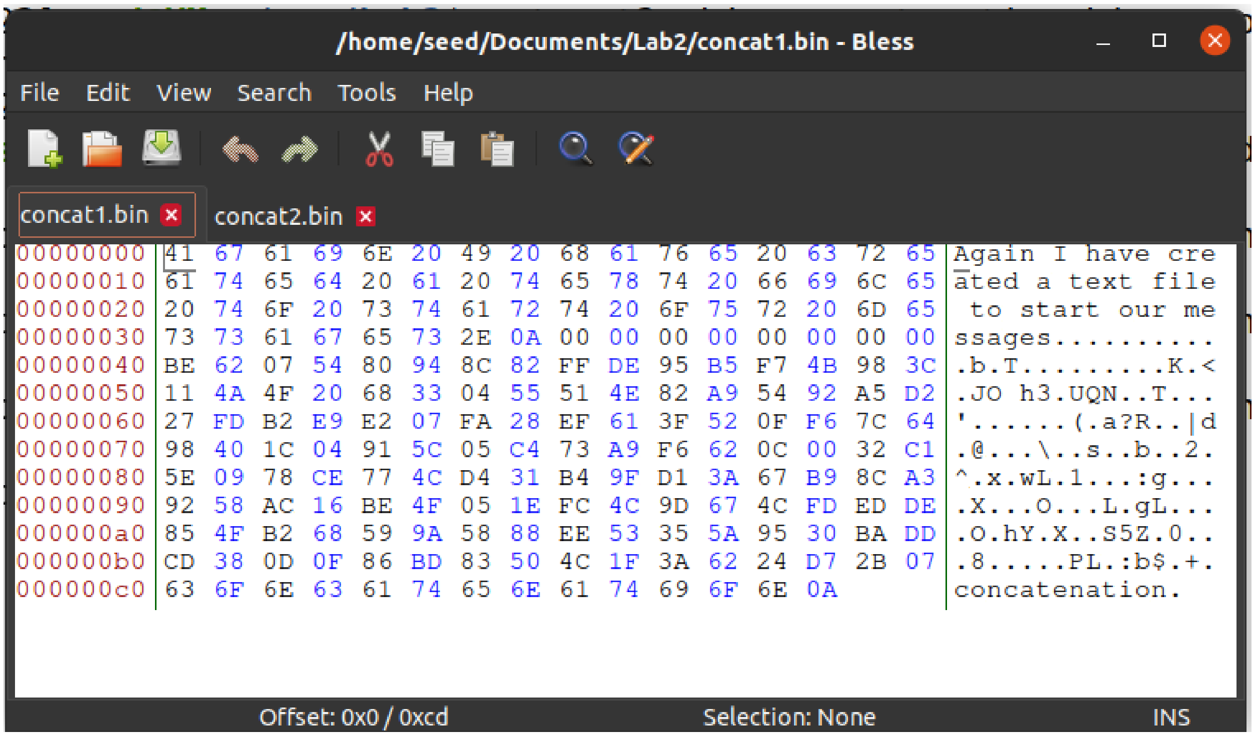

Fig 2.5 The file “concat1.bin” which has the prefix, bytes generated by md5collgen, and then the concatenation.

Links: [Table of Contents](#) | [References](#)

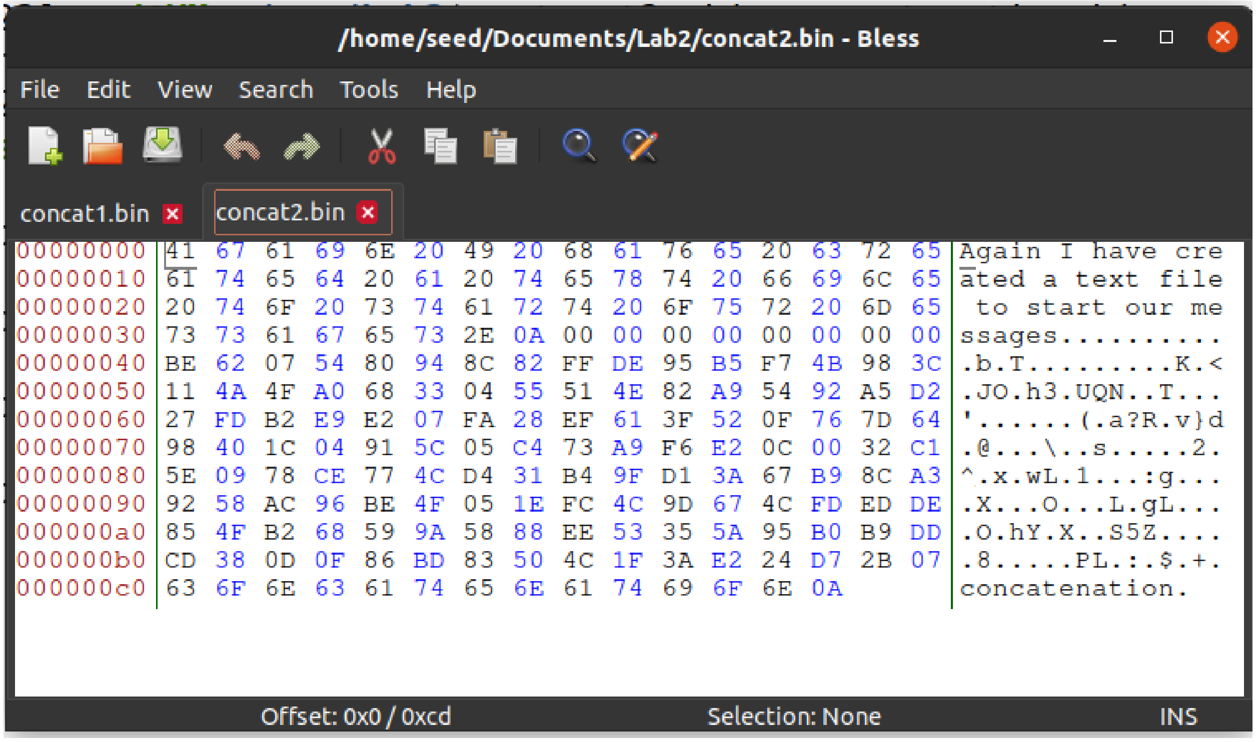

Fig 2.6 The file “concat2.bin” which has a similar layout to “concat1.bin” shown above.

The final step is to compare the hashes of these two files (Fig 2.7).

```
[10/08/22]seed@VM:~/.../Lab2$ md5sum concat1.bin concat2.bin
3f8c6fc0d88007e5aa9e08119bb8f179  concat1.bin
3f8c6fc0d88007e5aa9e08119bb8f179  concat2.bin
[10/08/22]seed@VM:~/.../Lab2$
```

Fig 2.7 The hashes of “concat1.bin” and “concat2.bin” respectively.

As we can see, this run of the experiment was a success. We were able to successively show that two different files with the same hash (“out1c.bin” and “out2c.bin”) still had identical hashes to each other when concatenated with a different message. In order to be sure of the results, I ran the experiment again using different output files from task 1 and using a different file as the concatenation (Fig 2.8 and Fig 2.9).

```
[10/08/22]seed@VM:~/.../Lab2$ md5sum out1b.bin out2b.bin
5457cf57d768750cf58453140cf3178a  out1b.bin
5457cf57d768750cf58453140cf3178a  out2b.bin
[10/08/22]seed@VM:~/.../Lab2$ cat out1b.bin prefix_a.txt > trial21.
bin
[10/08/22]seed@VM:~/.../Lab2$ cat out2b.bin prefix_a.txt > trial22.
bin
```

Fig 2.8 Confirming that out1b.bin and out2b.bin have the same hash, then concatenating them with the text file used in task 1.

```
[10/08/22]seed@VM:~/.../Lab2$ diff trial21.bin trial22.bin
2,3c2,3
< U����FL%w
           S�v��c�����s���      ���b�
                                     �r�A��]�\��S�p��o�-����$�T�,��
����E�m�����^t'�        tG�KO]M��\�
< �o�����4�kK�������(r�1This is an example of an MD5 message.
---
> U����FL%w
           S�v��c��j��s���      ���b�
                                     �r�A��]�\���S�p��o�-���/$�T�,�
�����E�m�����^�'�        tG�KO]M��\�
> �o�����4kK�������(r_�1This is an example of an MD5 message.
[10/08/22]seed@VM:~/.../Lab2$ md5sum trial21.bin trial22.bin
ba2017b5c01d55fa780346366b5dd376  trial21.bin
ba2017b5c01d55fa780346366b5dd376  trial22.bin
[10/08/22]seed@VM:~/.../Lab2$
```

Fig 2.9 Showing that even though the start of the two concated files for the second trial were different, they have the same hash.

Thus we can clearly see that for any two files N and M that have the same hash, we can concatenate another file T to each of them and the new files will still hold the property of having the same hash. We could even show this by concatening another file onto the already-concated one, or by choosing other files to put in the experiment. As long as the first part of the files has the same hash, it will produce the same IHV. Then as long as the following parts of the files are identical, they will also produce the same hash, since those later blocks start with the same IHV.

Task 3: Generating Two Executable Files with the Same MD5 Hash

Links: Table of Contents | References

In this task, we are given a program which prints each element of a 200-element array. By carefully choosing the elements in the array, we are supposed to generate two different programs which have the same MD5 hash. I pretty closely followed the guidance in the lab in order to accomplish this. I started by creating the program with the array all filled with 0s. I then compiled the program on the command line. However, when I opened the output file, there were several blocks of all 0s. So instead, I changed the array to be all 0x41 ('A') as suggested in the lab (Fig 3.1). Next, I opened the output file in bless to find the portion where the array information was stored.

```
[10/09/22]seed@VM:~/.../Lab2$ gcc ArrayProgram.c
[10/09/22]seed@VM:~/.../Lab2$ ./a.out
4141414141414141414141414141414141414141414141414141414141414141414
1414141414141414141414141414141414141414141414141414141414141414141
4141414141414141414141414141414141414141414141414141414141414141414
1414141414141414141414141414141414141414141414141414141414141414141
4141414141414141414141414141414141414141414141414141414141414141414
14141414141414141414141414141414141414141414141414141414141414141
```

Fig 3.1 The program printing out all 0s.

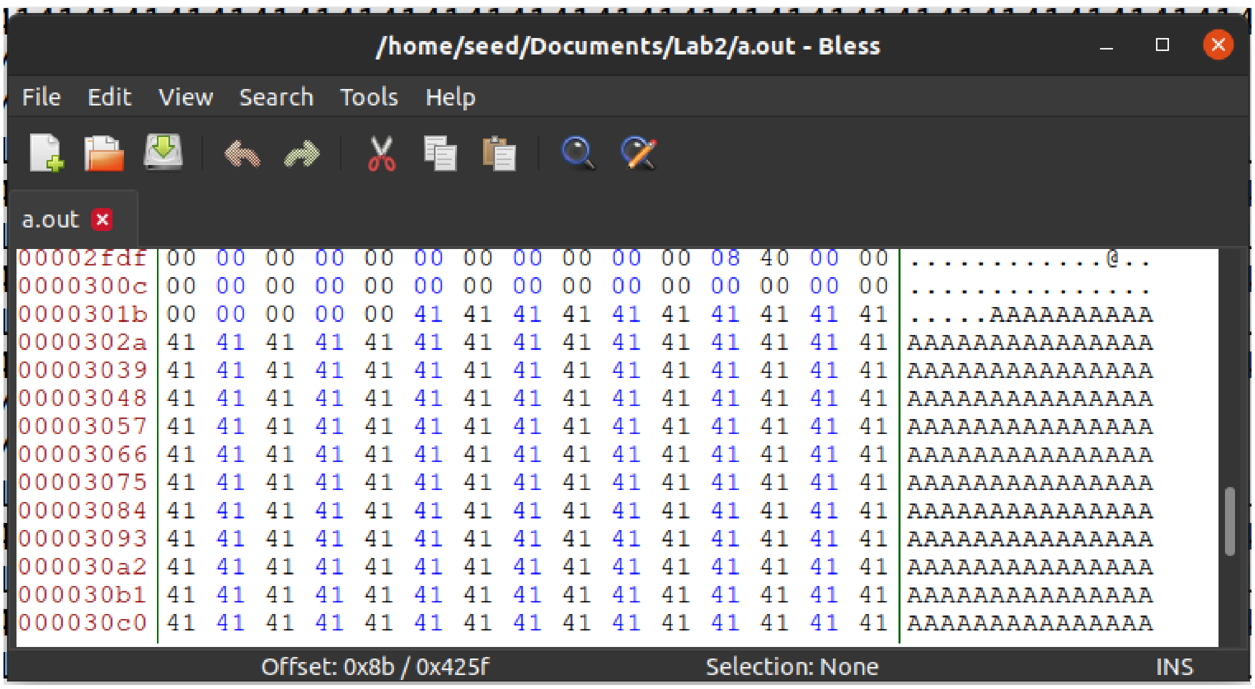

Fig 3.2 The portion of the file containing the array, which looks very similar to me studying for midterms.

Based on this information, I was able to find that the As started at byte 12320, which divided by 64 is 192.5. This meant that in order to make the prefix a multiple of 64, I would need to start at the 193rd chunk, which would be byte 12320 + .5*64 = 12352. I then cut the executable at this point (Fig 3.3).

```
[10/09/22]seed@VM:~/.../Lab2$ head -c 12352 a.out > prefix
[10/09/22]seed@VM:~/.../Lab2$ bless
```

Fig 3.3 Separating the first portion of the program.

This allowed me to run this first portion through the md5collgen program to generate two different blocks that would give the same hash value (Fig 3.4).

```
[10/09/22]seed@VM:~/.../Lab2$ md5collgen -p prefix -o headv1.bin he
adv2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'headv1.bin' and 'headv2.bin'
Using prefixfile: 'prefix'
Using initial value: 5f0fdc275f61d1036aeda0861ab918e6

Generating first block: .....................................
.....................................................
Generating second block: S11...
Running time: 66.024 s
```

Fig 3.4 Running the first portion of the program through the collision generator.

The resulting files “headv1.bin” and “headv2.bin” were each 12,480 bytes. I appended the tail of the original program to each of these heads (Fig 3.5).

```
[10/10/22]seed@VM:~/.../Lab2$ tail -c +12481 a.out > suffix
[10/10/22]seed@VM:~/.../Lab2$ cat headv1.bin suffix > programv1.out
[10/10/22]seed@VM:~/.../Lab2$ cat headv2.bin suffix > programv2.out
```

Fig 3.5 Concatenating the modified heads of the program with the original tail.

Finally, I configured the resulting file as an executable program (Fig 3.6).

Links: [Table of Contents] | [References]

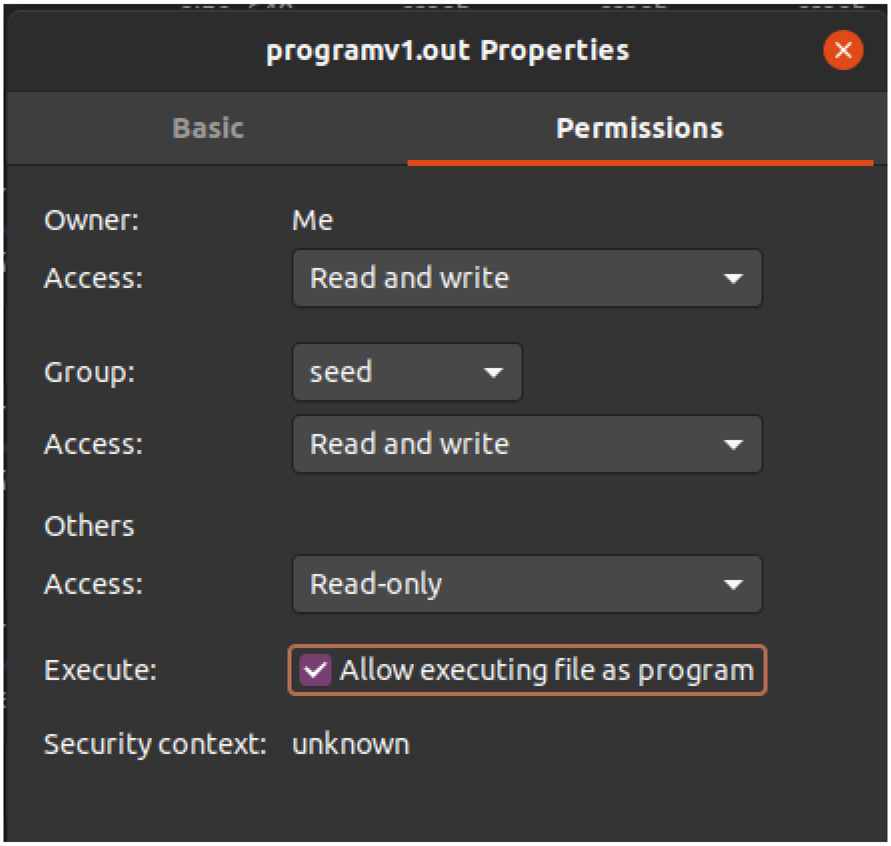

Fig 3.6 Allowing the concatenated program to be ran.

After running the two programs, I compared their outputs and found that 7 characters were different (Fig 3.7 and Fig 3.8). This is because md5collgen allowed me to generate two different ending blocks for the head of the program, which became the middle part of the contents of the array when I concatenated them. We can see in the output exactly where those characters were inserted, because they are both preceded and followed by the printout of ‘41’. Finally, I verified that their hashes were the same (Fig 3.9). As expected, they were, because of the property we proved in Task 2 that if two prefixes have the same hash, they will still have identical hashes with any appended suffix.

Links: [Table of Contents](#) | [References](#)

```
[10/10/22]seed@VM:~/.../Lab2$ ./programv1.out > program1out.txt
[10/10/22]seed@VM:~/.../Lab2$ ./programv2.out > program2out.txt
[10/10/22]seed@VM:~/.../Lab2$ differ program1out.txt program2out.tx
t
differ: command not found
[10/10/22]seed@VM:~/.../Lab2$ diff program1out.txt program2out.txt
1c1
< 414141414141414141414141414141414141414141414141414141414141419
7399699648f527bd9914f7b8a32fc2874110c1b4e7e86327a89719f13acdbb1640e
cd752ebc5f03b8a0294a784b6ece57dfe3135593fa4b8be648b35dc52982495431a
e32a1f20a0d3526c49462131e36c1f2e6c59dc4f4b6043c1e560bda2a7927f27b3e
758dd9e854120270af6cbba213aeaac22eddea341414141414141414141414141
41414141414141414141414141414141414141414141414141414141
---
> 414141414141414141414141414141414141414141414141414141414141419
7399699648f527bd9914f7b8a32fc2874190c1b4e7e86327a89719f13acdbb1640e
cd752ebc5f03b8202a4a784b6ece57dfe3135593facb8be648b35dc52982495431a
e32a1f20a0d3526c494621b1e36c1f2e6c59dc4f4b6043c1e560bda2a7927f27b3e
758dd1e854120270af6cbba213aeaa422eddea341414141414141414141414141
41414141414141414141414141414141414141414141414141414141
```

Fig 3.7 Saving the output of each version of the program and then confirming that they had different characters. Fig 3.8 below does a better job of highlighting the differences.

```
Removed Added

1  41414141414141414141414141414141414141414141414141414141414141419739969964 8f527bd9914f7b
2  8a32fc2874119 0c1b4e7e86327a89719f13acdbb1640ecd752ebc5f03b8a2029a4a784b6ece57dfe3135593fa4cb8b
3  e648b35dc5298249543 1ae32a1f20a0d3526c4946213b1e36c1f2e6c59dc4f4b6043c1e560bda2a7927f27b3e75
4  8dd91e854120270af6cbba213aeaac422eddea34141414141414141414141414141414141414141414141414141414
5  14141414141414141414141414141 v12
```

Fig 3.8 The differences in output viewed in a text comparator, which highlights the differences between version one (red) and version two (green) so that they are easier to visualize. I also added some line breaks in the text editor for readability.

```
[10/10/22]seed@VM:~/.../Lab2$ md5sum programv1.out programv2.out
00e41427c8a215274356fc8ccc8820bf  programv1.out
00e41427c8a215274356fc8ccc8820bf  programv2.out
```

Fig 3.9 Confirming that both versions of the program have different outputs.

Links: Table of Contents | References

Task 4: Making the Two Programs Behave Differently

This task asks us to create two different programs that share the same hash. One should run only benign instructions, while the other runs malicious instructions. Some guidance is given for how to accomplish this by having each version store a different set of characters in one array. Then, we can compare it to a second array and execute code based on whether that comparison is true or false. I will create my program based on similar logic. Each version of the program will store a different array. However, instead of comparing the entire array, I will check just one of the characters that is different between them. For the content of the arrays, I will use the output of the previous programs, since I know that pair of arrays in the execution file will hash to the same values. I opened the output in Binary Viewer (Ref 2), a free program similar to bless, since bless was having issues crashing when I tried to click anywhere. I just copied the output file into my shared folder, and then copied it from my shared folder onto my Windows desktop to use in Binary Viewer. Once I was able to view and search the file, I found two bytes that would be easily comparable within the program. Byte 78 in both versions was a distinct ASCII characters. In version 1, it was ')' (Fig 4.1), while in version 2 it was '*' (Fig 4.2).

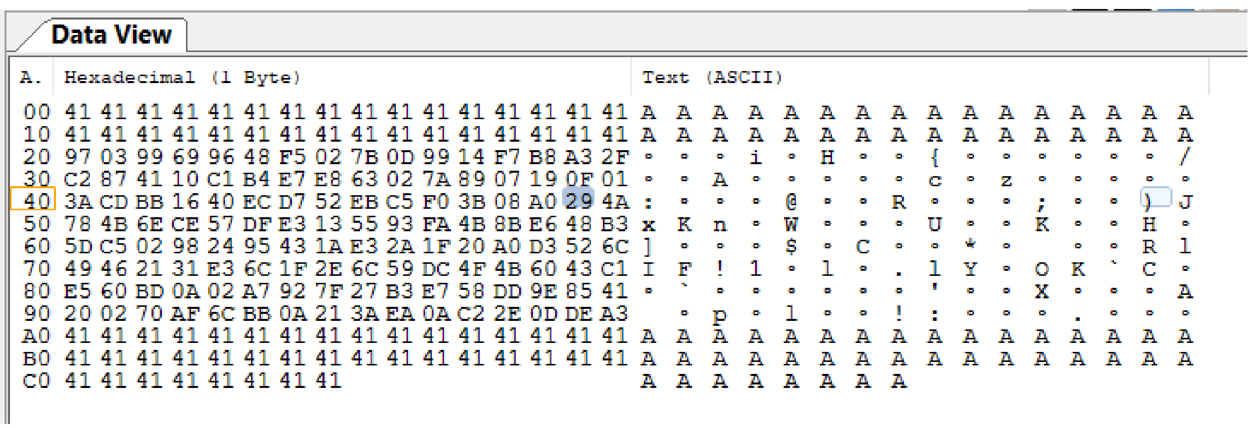

Fig 4.1 The array portion of the version1 of the program shown in Binary Viewer.

Links: [Table of Contents](#) | [References](#)

```
Data View
A.  Hexadecimal (1 Byte)                              Text (ASCII)
00  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   A A A A A A A A A A A A A A A A
10  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   A A A A A A A A A A A A A A A A
20  97 03 99 69 96 48 F5 02 7B 0D 99 14 F7 B8 A3 2F   ◦ ◦ ◦ i ◦ H ◦ ◦ { ◦ ◦ ◦ ◦ ◦ ◦ /
30  C2 87 41 90 C1 B4 E7 E8 63 02 7A 89 07 19 0F 01   ◦ ◦ A ◦ ◦ ◦ ◦ ◦ c ◦ z ◦ ◦ ◦ ◦ ◦
40  3A CD BB 16 40 EC D7 52 EB C5 F0 3B 08 20 2A 4A   : ◦ ◦ ◦ @ ◦ ◦ R ◦ ◦ ◦ ; ◦   * J
50  78 4B 6E CE 57 DF E3 13 55 93 FA CB 8B E6 48 B3   x K n ◦ W ◦ ◦ ◦ U ◦ ◦ ◦ ◦ ◦ H ◦
60  5D C5 02 98 24 95 43 1A E3 2A 1F 20 A0 D3 52 6C   ] ◦ ◦ ◦ $ ◦ C ◦ ◦ * ◦   ◦ ◦ R l
70  49 46 21 B1 E3 6C 1F 2E 6C 59 DC 4F 4B 60 43 C1   I F ! ◦ ◦ l ◦ . l Y ◦ O K ` C ◦
80  E5 60 BD 0A 02 A7 92 7F 27 B3 E7 58 DD 1E 85 41   ◦ ` ◦ ◦ ◦ ◦ ◦ ◦ ' ◦ ◦ X ◦ ◦ ◦ A
90  20 02 70 AF 6C BB 0A 21 3A EA 0A 42 2E 0D DE A3     ◦ p ◦ l ◦ ◦ ! : ◦ ◦ B . ◦ ◦ ◦
A0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   A A A A A A A A A A A A A A A A
B0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   A A A A A A A A A A A A A A A A
C0  41 41 41 41 41 41 41 41                           A A A A A A A A

File Name: programv2array.bin | Size: 200 Bytes | Address: 0000004E(Hex)/78(Dec) | Selection Size: 1 Bytes
```

Fig 4.2 The array portion of the version2 of the program shown in Binary Viewer, with the 78th byte highlighted.

This allowed me to now edit the original c program to execute different code based on the 78th character in the array, instead of just printing out the array regardless (Fig 4.3).

```
Open | DefinitelyGoodProgram.c ~/Documents/Lab2 | Save

#include <stdio.h>
unsigned char xyz[200] = {
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,(
};
int main()
{
if (xyz[77] == ')')
        printf("This is benevolent code.\n");
else
        printf("This is malicious code!\n");
}
```

Fig 4.3 The “definitely good” program that runs benevolent or malicious code based on the 77th byte of the array.

I thought the last step would be cutting the edited array portions of the program from task 3 with the new edited “definitely good program” executable file (Fig 4.4).

Links: [Table of Contents](#) | [References](#)

```
[10/10/22]seed@VM:~/.../Lab2$ gcc DefinitelyGoodProgram.c -o defgoo
d.out
[10/10/22]seed@VM:~/.../Lab2$ tail -c +12481 defgood.out > goodsuff
ix
[10/10/22]seed@VM:~/.../Lab2$ cat headv1.bin suffix > truegood.out
[10/10/22]seed@VM:~/.../Lab2$ cat headv2.bin suffix > fakegood.out
[10/10/22]seed@VM:~/.../Lab2$
```

Fig 4.4 Splicing the programs in what I thought was the correct way (but that actually led to incorrect behavior).

However, this led to the array being printed instead of the benevolent/malicious code being ran. So, I instead spliced only the first part of the array contents into the "defgood.out" file as shown below (Fig 4.5 and 4.6).

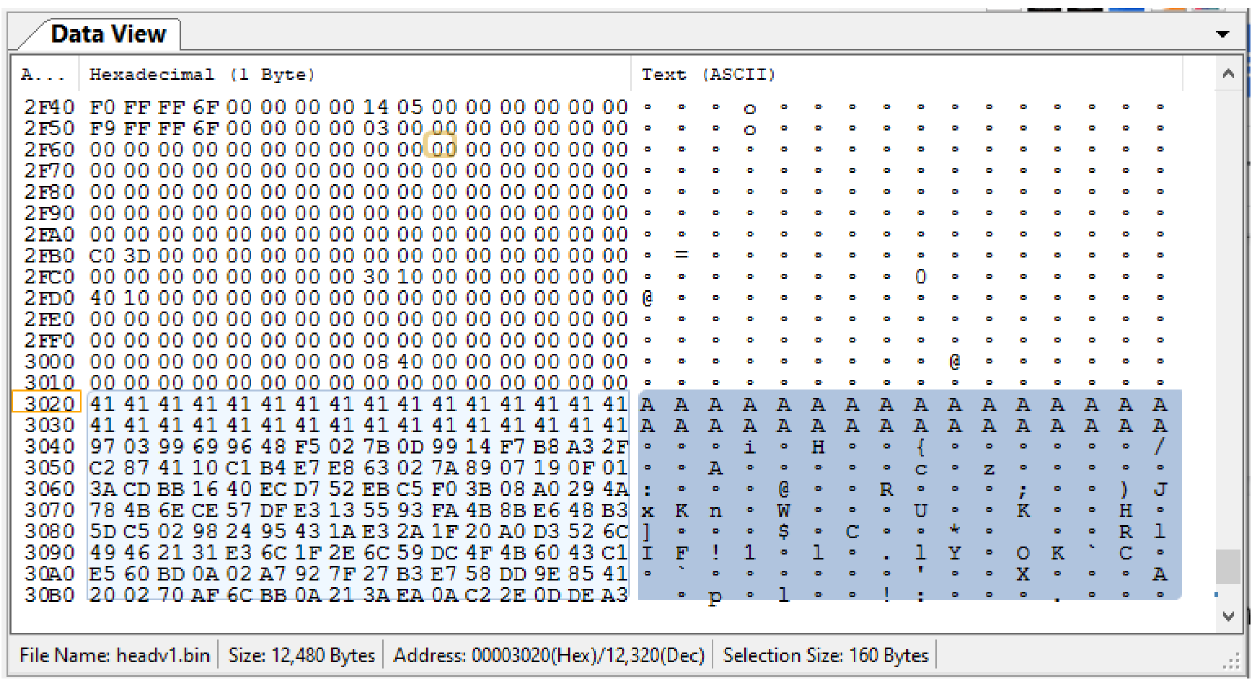

Fig 4.5 The bytes in headv1 that need to be removed and placed in the array

```
[10/10/22]seed@VM:~/.../Lab2$ tail -c 160 headv1.bin > arrayv1
[10/10/22]seed@VM:~/.../Lab2$ tail -c 160 headv2.bin > arrayv2
[10/10/22]seed@VM:~/.../Lab2$ head -c 12320 defgood.out > head
[10/10/22]seed@VM:~/.../Lab2$ tail -c +12480 defgood.out > tail
[10/10/22]seed@VM:~/.../Lab2$ tail -c +12479 defgood.out > tail
[10/10/22]seed@VM:~/.../Lab2$ cat head arrayv1 tail > goodv1.out
[10/10/22]seed@VM:~/.../Lab2$ cat head arrayv2 tail > goodv2.out
[10/10/22]seed@VM:~/.../Lab2$ ./goodv1.out
bash: ./goodv1.out: Permission denied
[10/10/22]seed@VM:~/.../Lab2$ ./goodv1.out
This is malicious code!
[10/10/22]seed@VM:~/.../Lab2$ ./goodv2.out
This is malicious code!
[10/10/22]seed@VM:~/.../Lab2$ tail -c +12481 defgood.out > tail
[10/10/22]seed@VM:~/.../Lab2$ cat head arrayv1 tail > goodv1.out
[10/10/22]seed@VM:~/.../Lab2$ cat head arrayv2 tail > goodv2.out
[10/10/22]seed@VM:~/.../Lab2$ ./goodv1.out
This is malicious code!
[10/10/22]seed@VM:~/.../Lab2$ ./goodv2.out
This is malicious code!
[10/10/22]seed@VM:~/.../Lab2$ diff goodv1.out goodv2.out
Binary files goodv1.out and goodv2.out differ
```

Fig 4.6 Splicing only the array content into the “defgood.out” program. We can see I had an issue at first with getting the correct number of bytes in the tail of the program.

Once I did that, it printed out the malicious code method for both versions, so something was still wrong. I realized that I was comparing the wrong byte; I had written “xyz[77]” when in fact I wanted to compare “xyz[78]”. So I updated that in the “DefinitelyGoodProgram.c” file, then recompiled (Fig 4.7), checked that the array location hadn’t changed (Fig 4.8), and respliced (Fig 4.9).

```
[10/10/22]seed@VM:~/.../Lab2$ gcc DefinitelyGoodProgram.c -o defgoo
da.out
[10/10/22]seed@VM:~/.../Lab2$ ./defgooda.out
This is malicious code!
```

Fig 4.7 Recompilinng the “DefinitelyGoodProgram.c” file after correcting the comparison byte.

Links: [Table of Contents] | [References]

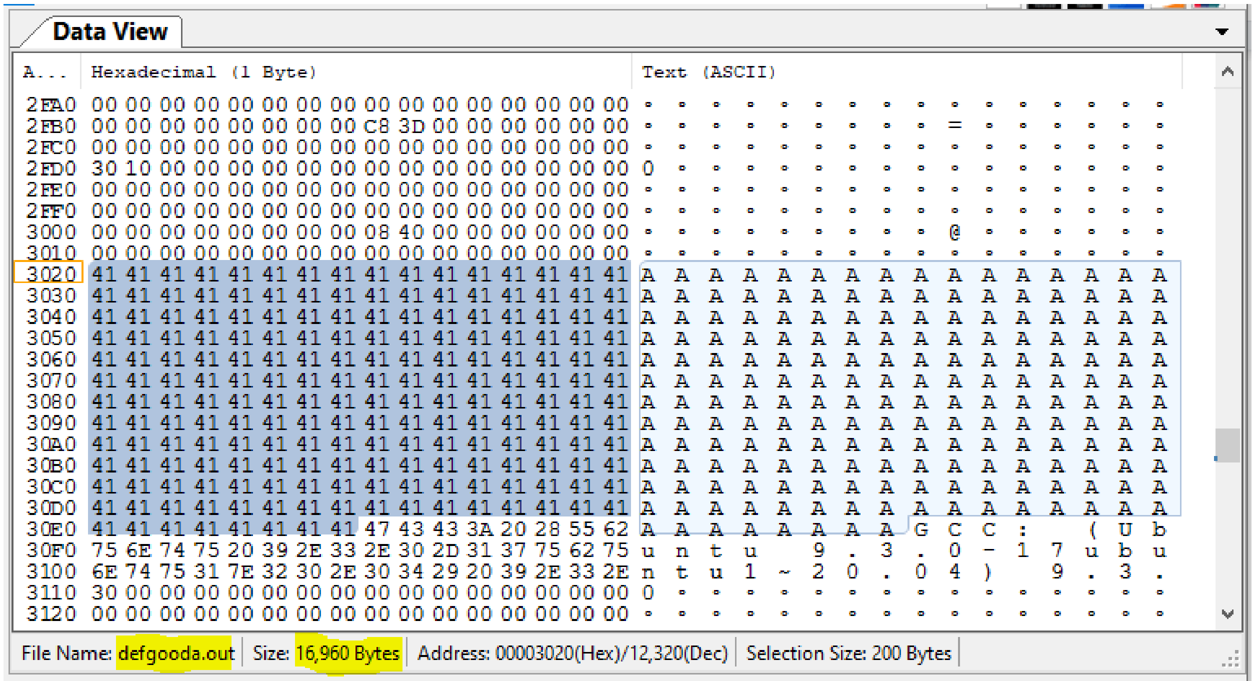

Fig 4.8 Confirming the file size and location of the array are the same for the recompiled version of the program.

```
[10/10/22]seed@VM:~/.../Lab2$ gcc DefinitelyGoodProgram.c -o defgoo
da.out
[10/10/22]seed@VM:~/.../Lab2$ ./defgooda.out
This is malicious code!
[10/10/22]seed@VM:~/.../Lab2$ head -c 12320 defgooda.out > heada
[10/10/22]seed@VM:~/.../Lab2$ tail -c +12481 defgooda.out > taila
[10/10/22]seed@VM:~/.../Lab2$ cat heada arrayv1 taila > goodv1a.out
[10/10/22]seed@VM:~/.../Lab2$ cat heada arrayv2 taila > goodv2a.out
[10/10/22]seed@VM:~/.../Lab2$ ./goodv1a.out
This is benevolent code.
[10/10/22]seed@VM:~/.../Lab2$ ./goodv2a.out
This is malicious code!
[10/10/22]seed@VM:~/.../Lab2$ md5sum goodv1a.out goodv2a.out
9b152c332c84e718efd12b1585dba198  goodv1a.out
50180e8f9849e1de70380cf162994962  goodv2a.out
```

Fig 4.9 Resplicing the code, and then checking the hash.

Unfortunately, once I checked the hash of the two programs, I found that it was not the same. I believe this is because changing the character comparison changes the compilation of the program and thus the hash of the program previous to the array. I could not find a way to circumvent this. If we focus on the C file instead of on the original file, then we run into the issue of formatting the array since all the elements are interspliced with commas. If compare one array to another, we still won't know what that array is before we run the hash, and we would then change the program by saving the array from the previous hash, catching us in the same loop.

References

1. Online Text Comparisons: https://countwordsfree.com/comparetexts
2. Binay Viewer Information: https://www.proxoft.com/BinaryViewer.aspx