It's not the end of the world: DarkComet misses by a mile

Reversing the DarkComet RAT's crypto- 3/13/2012

Jeff Edwards, Research Analyst, Arbor Networks ASERT

In this article, we will continue our series on reversing DDoS malware crypto systems. Previous subjects have included Armageddon, Khan (now believed to be a very close "cousin" of Dirt Jumper version 5), and PonyDOS. Today we'll be diving deep into the details of DarkComet's crypto. Over the last several months, we have encountered a large number of DarkComet samples, numbering well over a thousand. DarkComet is primarily a general purpose remote access trojan (RAT). It's capabilities support quite an extensive laundry list of mischief, including but not limited to key logging, web cam (and sound card) spying, deleting victim files, scanning ports, hijacking MSN sessions, etc.

Figure 1. Dark Comet's pretty logo

Of course the malware includes DDoS capabilities as well - hence our interest in reversing its communications so that we can keep tabs on whom the DarkComet botnets are attacking.   In fact, it is believed to have been used as a DDoS weapon by supporters of the Syrian regime against opposition forces in the recent Syrian uprisings; TrendMicro has a nice article /on this topic.

DarkComet has been studied by a number of researchers. In particular, in November 2011 Laura Aylward of Contextis published an excellent analysis [http://www.contextis.com/research/blog/darkcometrat/] of Dark Comet in which she described the basic cryptographic mechanism used by DarkComet bots to hide their communications; Laura's analysis saved us a considerable amount of time. It was also included in Curt Wilson's recent survey of modern DDoS weapons .

The DarkComet sample upon which we will primarily focus on today is 462,848 bytes in size and has an MD5 hash of `63f2ed5d2ee50e90cda809f2ac740244`. It happens to be an instance of DarkComet Version 4.2; however, the results presented here apply to most other versions of DarkComet as well.

When executed in a sandbox, we observed it connecting to a command & control (C&C) server at `newrat2.no-ip.org` on TCP port 1604. The RAT uses a raw TCP protocol to exchange information with its C&C; on the wire, the comms look something like this (modified and re-encrypted to protect some of our sensitive sandbox information):

```
C&C:
155CAD31A61F
Bot:
0F5DAB3EB308
C&C:
1B7D8D3BBF14C6B619480C265C2F4664F9DCB878EA7DFC6F2637
Bot:
35769F079329B4E04603496A432E5A7CFC90A477F478F07A3826A1B436AB92852B685636
F72B52C56D70434D7691F3307D637118B869586A1D19FD15B8C6AE14F8F8C57EFAFCCC09
964E8EE8EED553886AB188665F1AB96586F4F2581C093E75DCF2A8ADC817558BF3452344
0CDBE43CA4C05AC6E8D90D00F35BE795A44AE0E2EDE36C061EAEBD754461F680DBD9893A
CF6211698AF22B0BBB92A9B47363AE86E69A08C29DD3DBA59D287E4A0E12664B312A81C0
E9FE4D6E538AB5CC8952CCB372869F57D168CE8ABB52B8D7F8E78547A5EB009931735868
```

```
ADEC6BA2B73A94C7A9A6784B1A81C58CF746D384B645DD02D4616479A055420DADEF0458
658A33EEA62BF7F12ABF1C0E00CB6B971869FBC275A3270E8DEBFA20E53E8C3BC6CA2744
A88897E0B16FBBDCAA731B93A72D75FF6DC297
Bot:
KEEPALIVE144357
Bot:
C: KEEPALIVE160360
C&C:
S: KeepAlive|27120274
Bot:
C: KEEPALIVE176363
Bot:
C: KEEPALIVE192366
C&C:
S: KeepAlive|27160288
```

Figure 2. Example of DarkComet's encrypted comms

These communications are consistent with those reported by Contextis in their DarkComet report. It certainly looks like an initial "phone home" exchange of information, after which the bot and C&C send periodic "Keep Alive" messages to each other. Besides being encrypted, this protocol is somewhat unusual in that the C&C sends the first payload; it is much more common for the bot to send the first payload.

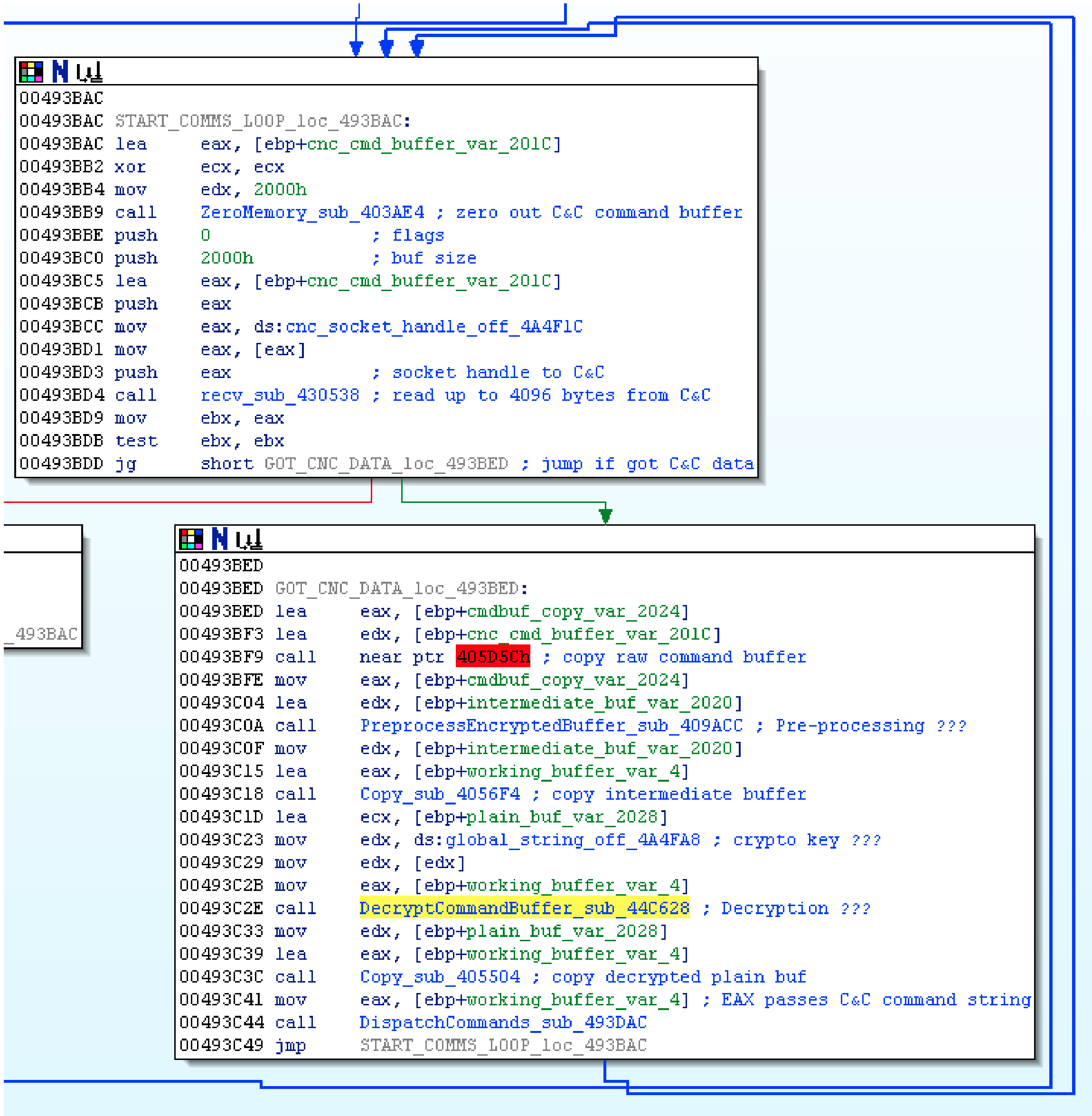
So in order to develop a tracker that impersonates a DarkComet bot so as to snoop on DDoS attacks, we need to reverse the malware's crypto system and write decryption and encryption routines in Python. Let's start reversing by loading a process memory dump of the running bot in IDA Pro. We'll then start poking around looking for routines that might implement the phone home protocol. Since DarkComet clearly uses raw TCP for communication (as opposed to, say, HTTP), we'll focus on finding WinSock2 calls such as `socket()`, `connect()`, `send()`, and `recv()`.

Well, it turns out that the bot is riddled with vast numbers of WinSock2 calls; not surprising, since DarkComet has a great deal of RAT functions that require network communication. So to narrow down on the actual bot-C&C comms loop, we locate the lengthy list of command strings, such as `KeylogOn`, `GetOfflineLogs`, `WEBCAMLIVE`, `GetMsnList`, `DDOSHTTPFLOOD`, etc. In particular, we note that all these command strings are referenced from the same function. Furthermore, this function is structured as a very long sequence of `if`-`else` statements that compare each of these command strings against the same buffer. Even better, there is only a single caller of this function. Hmmm, that certainly sounds like the bot's primary command dispatch routine; we'll call it `DispatchCommands_sub_493DAC()`.

Checking out the caller function, we see that it operates in a loop. On each iteration through the loop, it basically performs the following actions:

1. Calls `recv()` to read network traffic into a buffer;
2. Performs some copies and operations on this buffer to produce an intermediate buffer;
3. Performs an operation (decryption perhaps?) on the intermediate buffer and a global string to produce a final buffer;
3. Passes the final buffer to the aforementioned `DispatchCommands_sub_493DAC()` function;

Yes, this sounds like the main comms loop for which we are looking; we'll name this caller function `MainCommsLoop_sub_493A30()`, and focus our attention on the aforementioned loop:



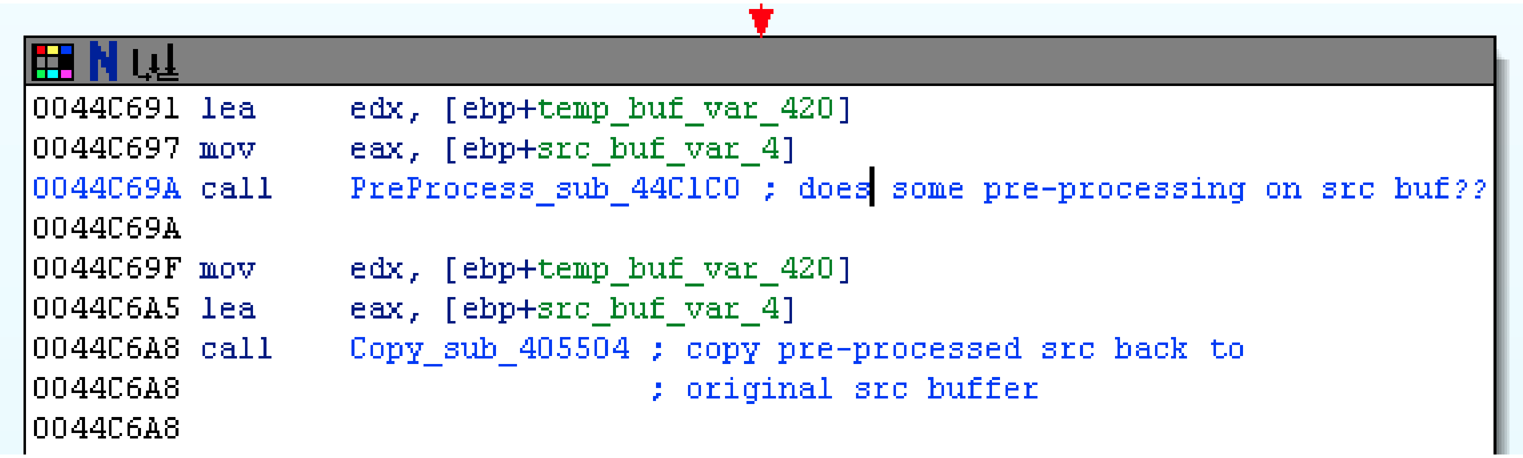
```
00493BAC
00493BAC START_COMMS_LOOP_loc_493BAC:
00493BAC lea     eax, [ebp+cnc_cmd_buffer_var_201C]
00493BB2 xor     ecx, ecx
00493BB4 mov     edx, 2000h
00493BB9 call    ZeroMemory_sub_403AE4 ; zero out C&C command buffer
00493BBE push    0                ; flags
00493BC0 push    2000h            ; buf size
00493BC5 lea     eax, [ebp+cnc_cmd_buffer_var_201C]
00493BCB push    eax
00493BCC mov     eax, ds:cnc_socket_handle_off_4A4F1C
00493BD1 mov     eax, [eax]
00493BD3 push    eax              ; socket handle to C&C
00493BD4 call    recv_sub_430538 ; read up to 4096 bytes from C&C
00493BD9 mov     ebx, eax
00493BDB test    ebx, ebx
00493BDD jg      short GOT_CNC_DATA_loc_493BED ; jump if got C&C data
```

```
00493BED
00493BED GOT_CNC_DATA_loc_493BED:
00493BED lea     eax, [ebp+cmdbuf_copy_var_2024]
00493BF3 lea     edx, [ebp+cnc_cmd_buffer_var_201C]
00493BF9 call    near ptr 405D5Ch ; copy raw command buffer
00493BFE mov     eax, [ebp+cmdbuf_copy_var_2024]
00493C04 lea     edx, [ebp+intermediate_buf_var_2020]
00493C0A call    PreprocessEncryptedBuffer_sub_409ACC ; Pre-processing ???
00493C0F mov     edx, [ebp+intermediate_buf_var_2020]
00493C15 lea     eax, [ebp+working_buffer_var_4]
00493C18 call    Copy_sub_4056F4 ; copy intermediate buffer
00493C1D lea     ecx, [ebp+plain_buf_var_2028]
00493C23 mov     edx, ds:global_string_off_4A4FA8 ; crypto key ???
00493C29 mov     edx, [edx]
00493C2B mov     eax, [ebp+working_buffer_var_4]
00493C2E call    DecryptCommandBuffer_sub_44C628 ; Decryption ???
00493C33 mov     edx, [ebp+plain_buf_var_2028]
00493C39 lea     eax, [ebp+working_buffer_var_4]
00493C3C call    Copy_sub_405504 ; copy decrypted plain buf
00493C41 mov     eax, [ebp+working_buffer_var_4] ; EAX passes C&C command string
00493C44 call    DispatchCommands_sub_493DAC
00493C49 jmp     START_COMMS_LOOP_loc_493BAC
```

Figure 3. Function `MainCommsLoop_sub_493A30()`

It definitely looks like a great candidate for the decryption operation. It follows the general structure that is quite common among bot families that encrypt their comms; namely, a pre-processing operation applied to a buffer, followed by the actual decryption step. In particular, one strong clue is that the (assumed) decryption step takes a third argument which, in this case, is a reference to a global string - very likely to be the decryption key string!

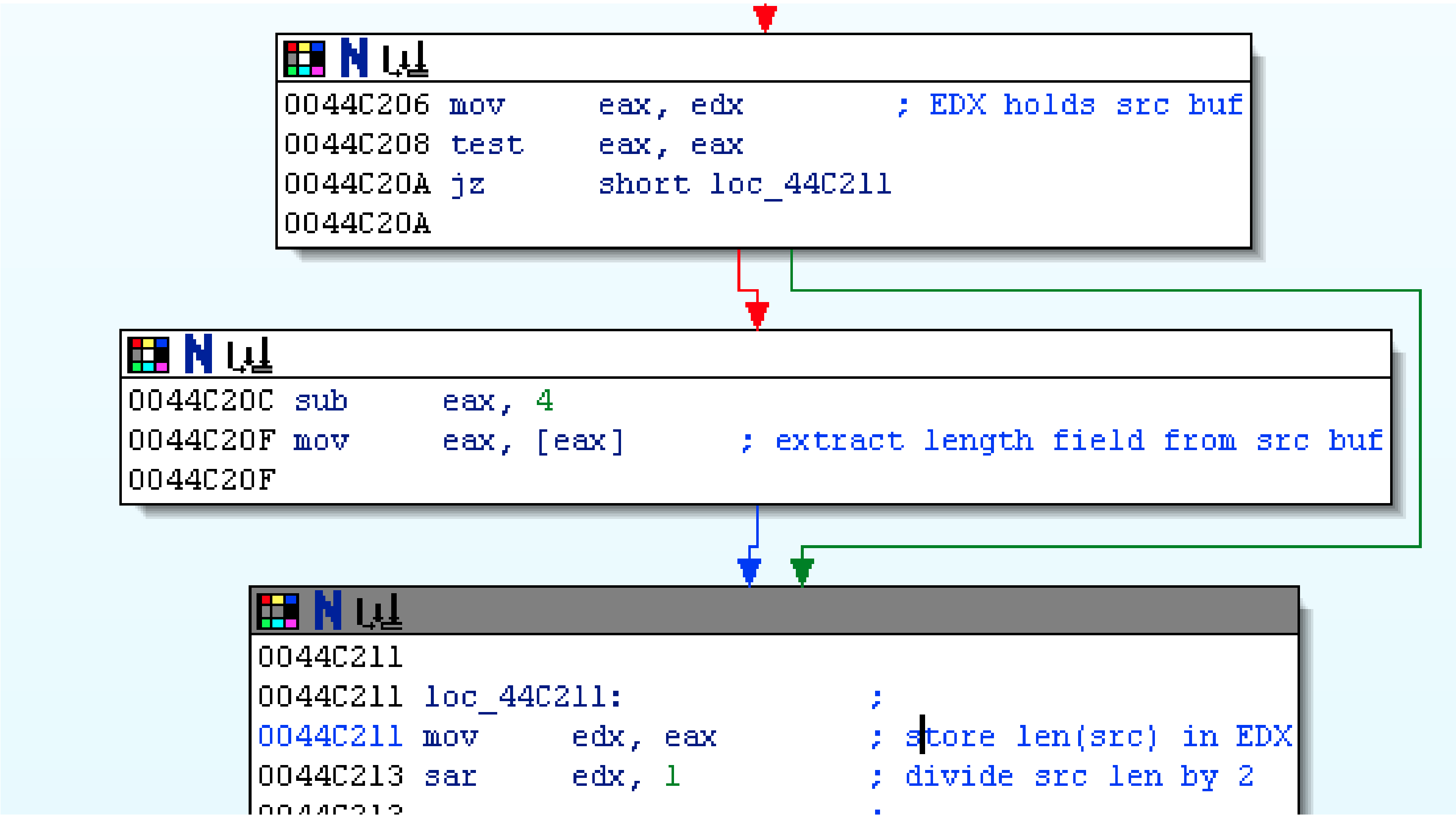
So first let's see what our (tentatively named) `DecryptCommandBuffer_sub_44C628()` function looks like. DarkComet being a Delphi-based bot, the decryption function is passed the source (encrypted) buffer in `EAX`, the (presumed) crypto key in `EDX`, and an output string buffer in `ECX`. After checking to make sure neither the source nor key strings are empty, the function gets down to business. The first substantive operation is to pass the raw (encrypted) source buffer `src_buf_var_4` via `EAX`, along with an output buffer `temp_buf_var_420` via `EDX`, to function `sub_44C1C0()`; the output buffer is then copied back into the original source buffer `src_buf_var_4`:



```
0044C691 lea     edx, [ebp+temp_buf_var_420]
0044C697 mov     eax, [ebp+src_buf_var_4]
0044C69A call    PreProcess_sub_44C1C0 ; does some pre-processing on src buf??
0044C69A
0044C69F mov     edx, [ebp+temp_buf_var_420]
0044C6A5 lea     eax, [ebp+src_buf_var_4]
0044C6A8 call    Copy_sub_405504 ; copy pre-processed src back to
0044C6A8                         ; original src buffer
0044C6A8
```

Figure 4. Function `DecryptCommandBuffer_sub_44C628()`

So `sub_44C1C0()` seems like it might be doing some pre-processing on the encrypted source buffer; let's see what kind of pre-processing it is doing. Skipping past the obligatory checks for empty source buffers, etc., we arrive at some code that loops over the source buffer, referenced by `src_buf_var_4`; however, it makes only one loop iteration for every two bytes in `src_buf_var_4`. This is accomplished by extracting the `DWORD` just in front of the source string and shifting it one bit to the right, in order to calculate the number of *pairs* of source characters:



```
0044C206 mov     eax, edx          ; EDX holds src buf
0044C208 test    eax, eax
0044C20A jz      short loc_44C211
0044C20A
```

```
0044C20C sub     eax, 4
0044C20F mov     eax, [eax]        ; extract length field from src buf
0044C20F
```

```
0044C211
0044C211 loc_44C211:               ;
0044C211 mov     edx, eax          ; store len(src) in EDX
0044C213 sar     edx, 1            ; divide src len by 2
```

Figure 5. Function `PreProcess_sub_44C1C0()`

This works because in Delphi, the `AnsiString` class stores its length at an offset of 4 bytes in front of the first actual byte of string content:

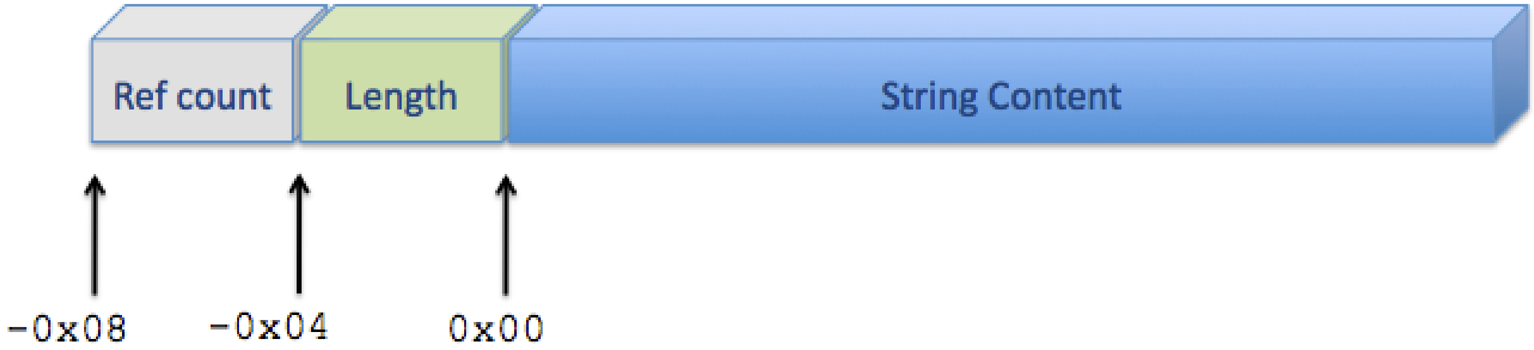
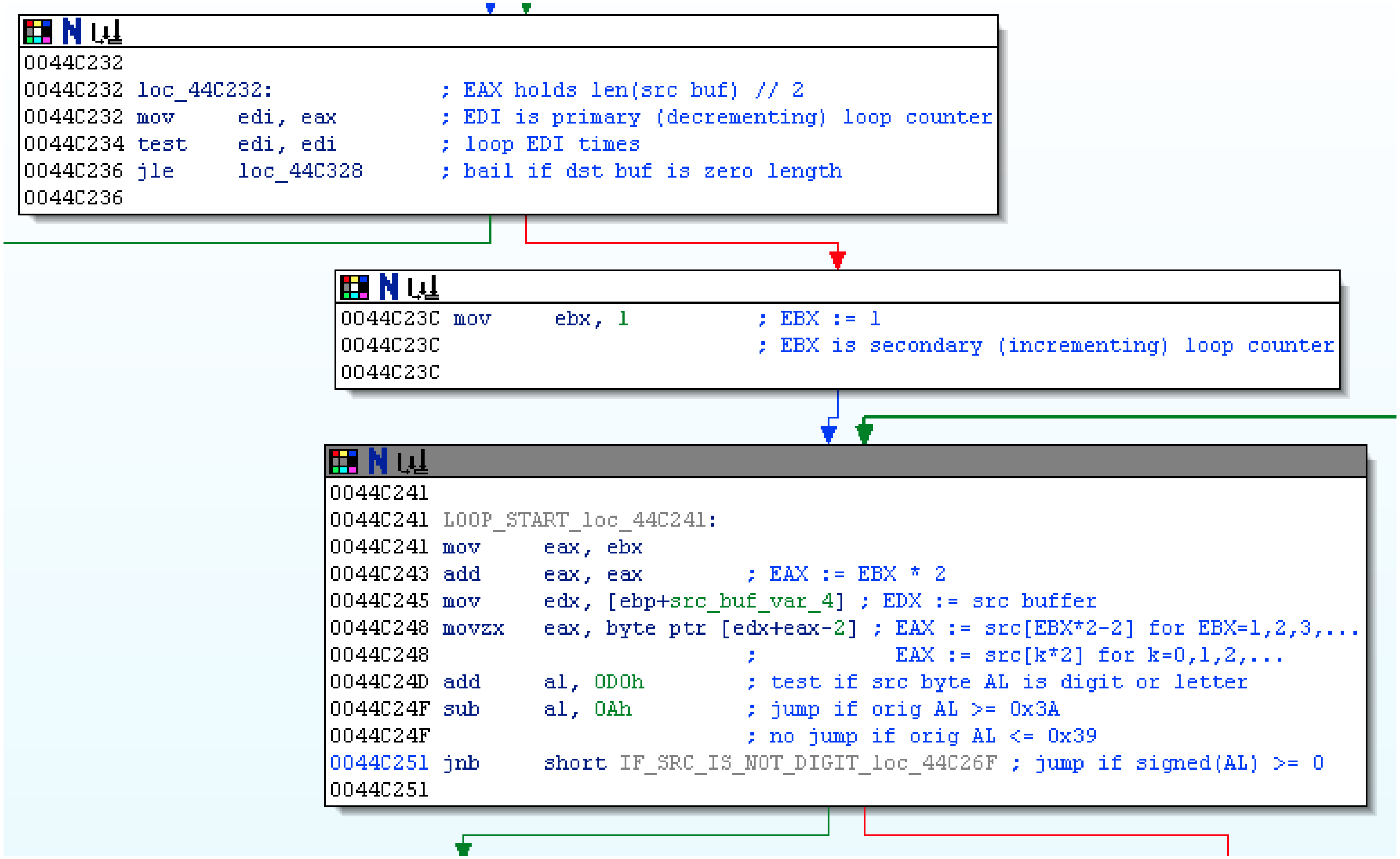


Figure 6. Structure of a Delphi `AnsiString`

For example, in the case of the initial encrypted payload received by the bot from the C&C, `155CAD31A61F` , the length of the source buffer is 12, so the code will make only 6 iterations through the loop. On each iteration of the loop, DarkComet will process a pair of two source bytes to yield one output byte.

The first operation inside the loop is to test whether or not the value of the first source byte in the pair is greater than `0x39`, and branch accordingly. After using the one-based index `EBX` to pull out the first of the two source bytes in the pair, it adds `0xD0`, subtracts `0x0A`, and then tests whether the resulting value is greater than or equal to zero. Since it is operating on the 8-bit register `AL`, the result is that source bytes with values of `0x3A` or greater will be processed by one branch, and those with values of `0x39` and less will be processed by a second branch:



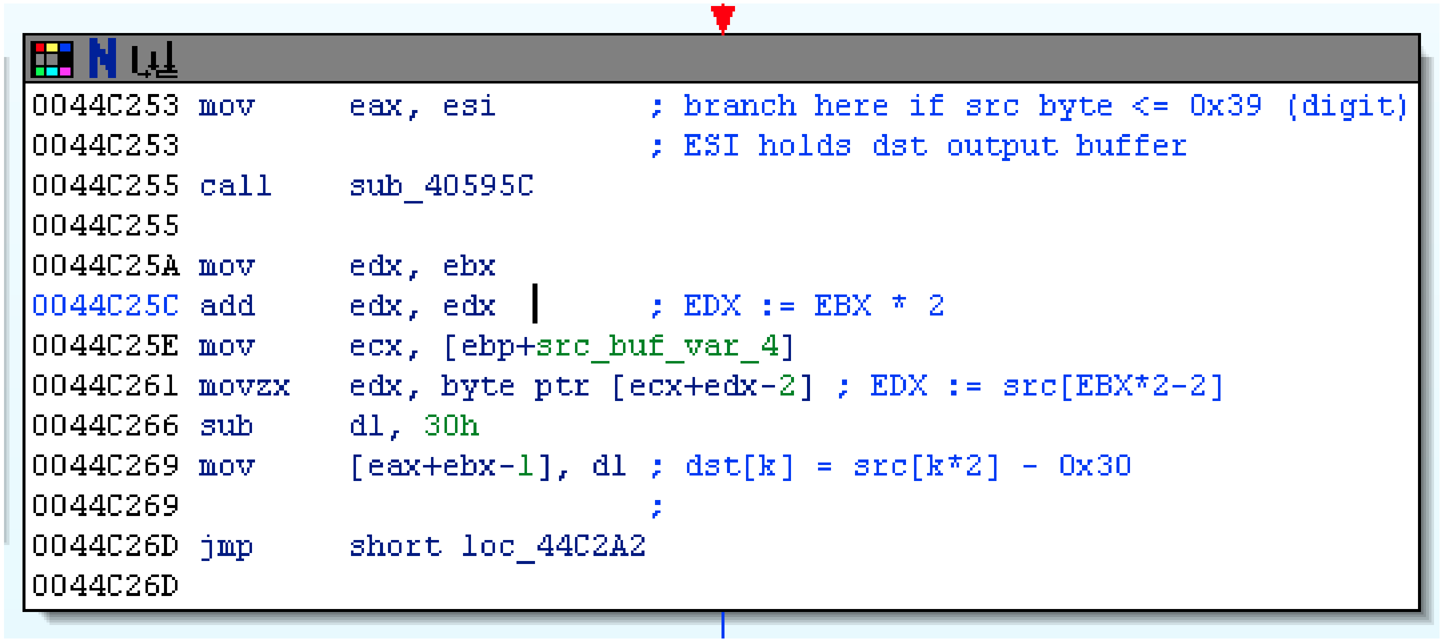
```
0044C232
0044C232 loc_44C232:                 ; EAX holds len(src buf) // 2
0044C232 mov     edi, eax            ; EDI is primary (decrementing) loop counter
0044C234 test    edi, edi            ; loop EDI times
0044C236 jle     loc_44C328          ; bail if dst buf is zero length
0044C236
```

```
0044C23C mov     ebx, 1              ; EBX := 1
0044C23C                             ; EBX is secondary (incrementing) loop counter
0044C23C
```

```
0044C241
0044C241 LOOP_START_loc_44C241:
0044C241 mov     eax, ebx
0044C243 add     eax, eax            ; EAX := EBX * 2
0044C245 mov     edx, [ebp+src_buf_var_4] ; EDX := src buffer
0044C248 movzx   eax, byte ptr [edx+eax-2] ; EAX := src[EBX*2-2] for EBX=1,2,3,...
0044C248                             ;            EAX := src[k*2] for k=0,1,2,...
0044C24D add     al, 0D0h            ; test if src byte AL is digit or letter
0044C24F sub     al, 0Ah             ; jump if orig AL >= 0x3A
0044C24F                             ; no jump if orig AL <= 0x39
0044C251 jnb     short IF_SRC_IS_NOT_DIGIT_loc_44C26F ; jump if signed(AL) >= 0
0044C251
```

Figure 7. Function `PreProcess_sub_44C1C0()`

If the first source byte in the pair has value `0x39` or less, the bot will subtract `0x30` from it and save the result to the current index within the output buffer:

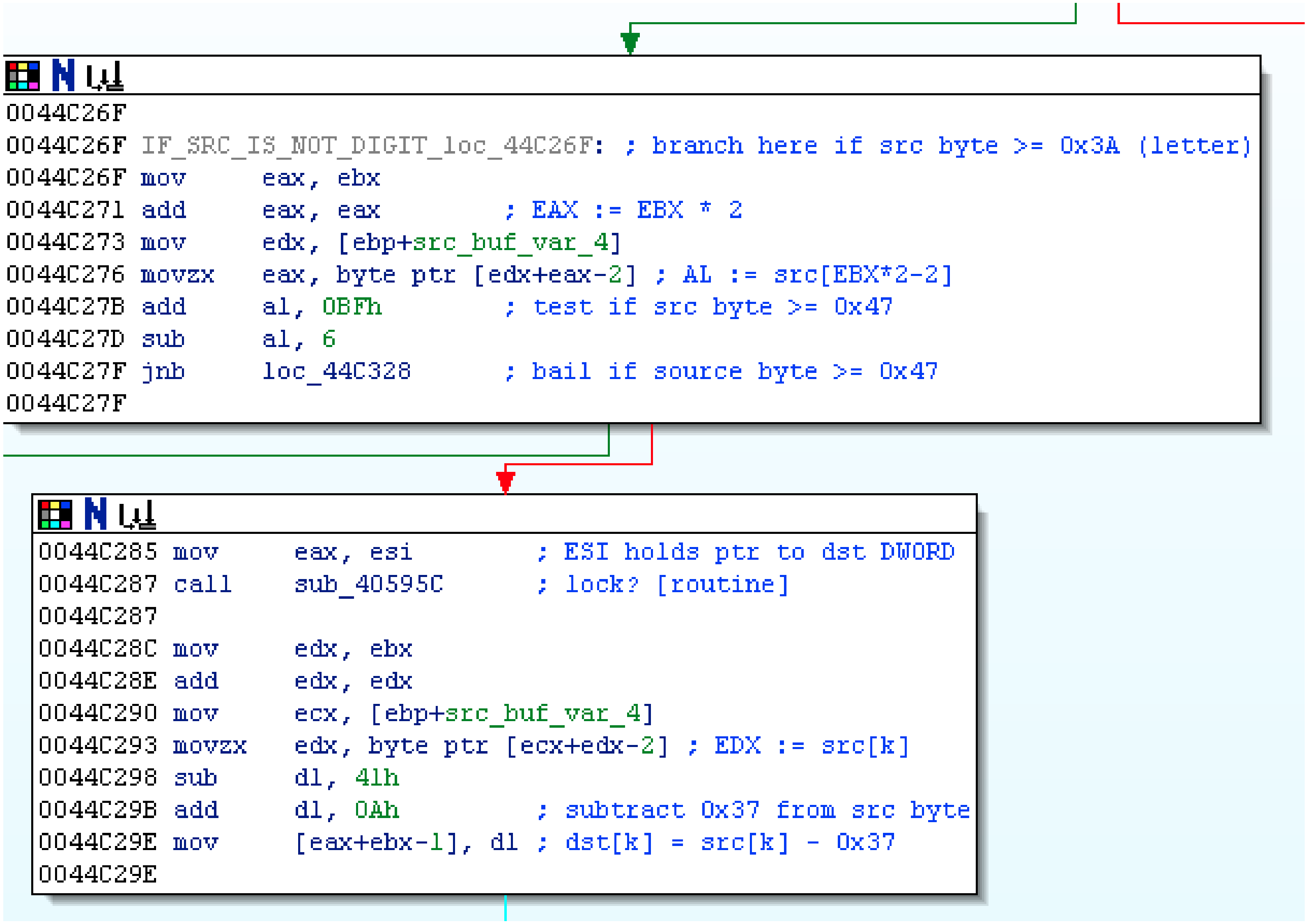


```
0044C253 mov     eax, esi          ; branch here if src byte <= 0x39 (digit)
0044C253                           ; ESI holds dst output buffer
0044C255 call    sub_40595C
0044C255
0044C25A mov     edx, ebx
0044C25C add     edx, edx          ; EDX := EBX * 2
0044C25E mov     ecx, [ebp+src_buf_var_4]
0044C261 movzx   edx, byte ptr [ecx+edx-2] ; EDX := src[EBX*2-2]
0044C266 sub     dl, 30h
0044C269 mov     [eax+ebx-1], dl   ; dst[k] = src[k*2] - 0x30
0044C269                           ;
0044C26D jmp     short loc_44C2A2
0044C26D
```

Figure 8. Function `PreProcess_sub_44C1C0()`

In other words, it will convert the ASCII representations (`0x30`, `0x31`, ..., `0x39`) of the digits `0` through `9` into their equivalent integer representations (`0x00`, `0x01`, ..., `0x09`).

The second branch performs a similar operation: it first tests to make sure that the value of the source byte is not `0x47` or greater (in which case it will immediately bail out of the loop and jump to the end of the `PreProcess_sub_44C1C0()` function.) It will then subtract `0x37` from the source byte and save the result into the current index within the output buffer:



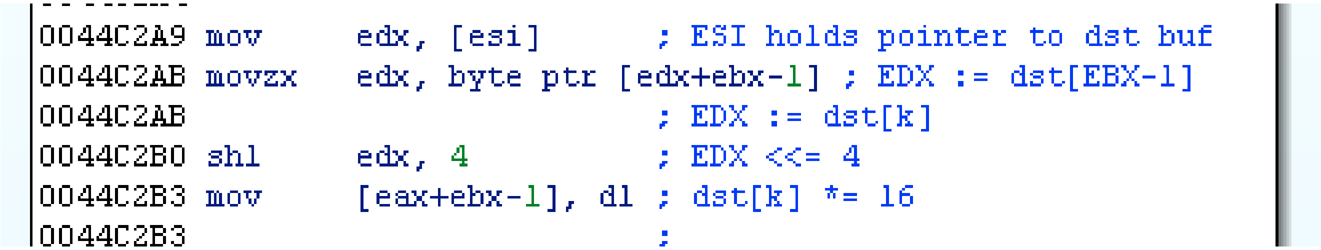
```
0044C26F
0044C26F IF_SRC_IS_NOT_DIGIT_loc_44C26F: ; branch here if src byte >= 0x3A (letter)
0044C26F mov     eax, ebx
0044C271 add     eax, eax          ; EAX := EBX * 2
0044C273 mov     edx, [ebp+src_buf_var_4]
0044C276 movzx   eax, byte ptr [edx+eax-2] ; AL := src[EBX*2-2]
0044C27B add     al, 0BFh          ; test if src byte >= 0x47
0044C27D sub     al, 6
0044C27F jnb     loc_44C328        ; bail if source byte >= 0x47
0044C27F
```

```
0044C285 mov     eax, esi          ; ESI holds ptr to dst DWORD
0044C287 call    sub_40595C        ; lock? [routine]
0044C287
0044C28C mov     edx, ebx
0044C28E add     edx, edx
0044C290 mov     ecx, [ebp+src_buf_var_4]
0044C293 movzx   edx, byte ptr [ecx+edx-2] ; EDX := src[k]
0044C298 sub     dl, 41h
0044C29B add     dl, 0Ah           ; subtract 0x37 from src byte
0044C29E mov     [eax+ebx-1], dl   ; dst[k] = src[k] - 0x37
0044C29E
```

Figure 9. Function `PreProcess_sub_44C1C0()`

In other words, it will convert the ASCII representations (`0x41`, `0x42`, ..., `0x46`) of the upper-case letters `A` through `F` into their equivalent hexadecimal representations (`0x0A`, `0x0B`, ..., `0x0F`).

The two branches (for handling digits and upper-case `A` through `F`) will then re-join, and the resulting integer/hexadecimal representation of the first source byte will be left-shifted by four (thus multiplying it by 16):



```
0044C2A9 mov     edx, [esi]       ; ESI holds pointer to dst buf
0044C2AB movzx   edx, byte ptr [edx+ebx-1] ; EDX := dst[EBX-1]
0044C2AB                          ; EDX := dst[k]
0044C2B0 shl     edx, 4           ; EDX <<= 4
0044C2B3 mov     [eax+ebx-1], dl  ; dst[k] *= 16
0044C2B3                          ;
```

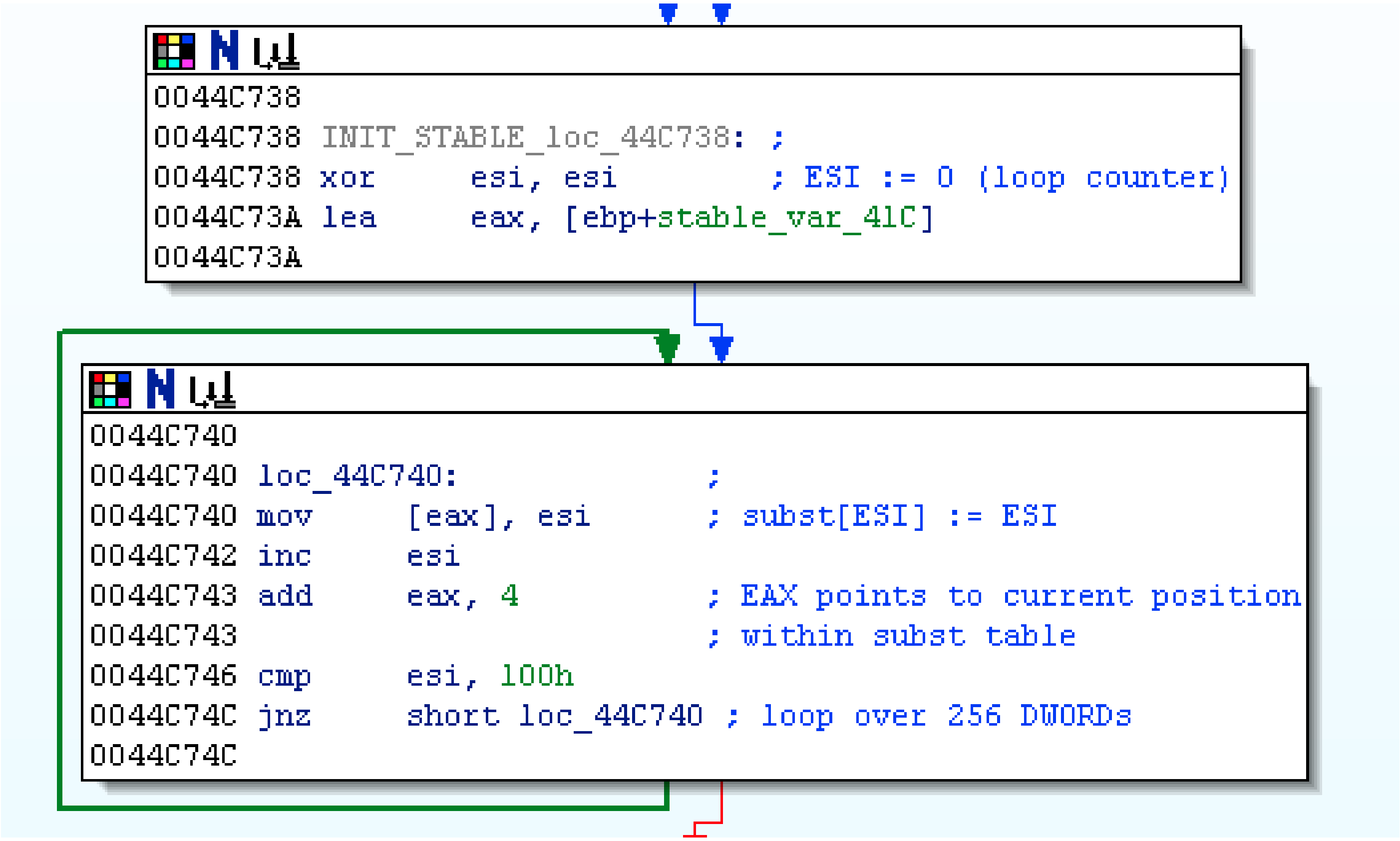
Figure 10. Function `PreProcess_sub_44C1C0()`

At this point, it is pretty clear what is going on. The `PreProcess_sub_44C1C0()` function is converting the ASCII representation of the source string of bytes into the equivalent hexadecimal representation. This conjecture is confirmed upon inspection of the remaining portion of the loop, which applies the same ASCII-to-hex operation on the second byte of each pair of source bytes, and adds the result to the left-shifted output from the first byte of the pair. So at the end of the day, the first line of raw encrypted source payload from the C&C is pre-processed from the 12-character ASCII string `155CAD31A61F` to its equivalent sequence of six hexadecimal bytes `0x15 0x5C 0xAD 0x31 0xA6 0x1F`, as follows:

src index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
src (ASCII)	1	5	5	C	A	D	3	1	A	6	1	F
src (raw)	0x31	0x35	0x35	0x43	0x41	0x44	0x33	0x31	0x41	0x36	0x31	0x46
src (hex)	0x01	0x05	0x05	0x0C	0x0A	0x0D	0x03	0x01	0x0A	0x06	0x01	0x0F
shifted	0x10		0x50		0xA0		0x30		0xA0		0x10	
dst	0x15		0x5C		0xAD		0x31		0xA6		0x1F	

Figure 11. ASCII to Integer Conversion

So we will rename this function as `Integerize_sub_44C1C0()`, and head back to the main `DecryptCommandBuffer_sub_44C628()` function to continue reversing the crypto algorithm. After the raw source buffer has been converted from ASCII form to integer form, the next substantive code block initializes a 256-element array `stable_var_41C`:



```
0044C738
0044C738 INIT_STABLE_loc_44C738: ;
0044C738 xor     esi, esi         ; ESI := 0 (loop counter)
0044C73A lea     eax, [ebp+stable_var_41C]
0044C73A
```

```
0044C740
0044C740 loc_44C740:            ;
0044C740 mov     [eax], esi     ; subst[ESI] := ESI
0044C742 inc     esi
0044C743 add     eax, 4         ; EAX points to current position
0044C743                        ; within subst table
0044C746 cmp     esi, 100h
0044C74C jnz     short loc_44C740 ; loop over 256 DWORDs
0044C74C
```

Figure 12. Function `DecryptCommandBuffer_sub_44C628()`

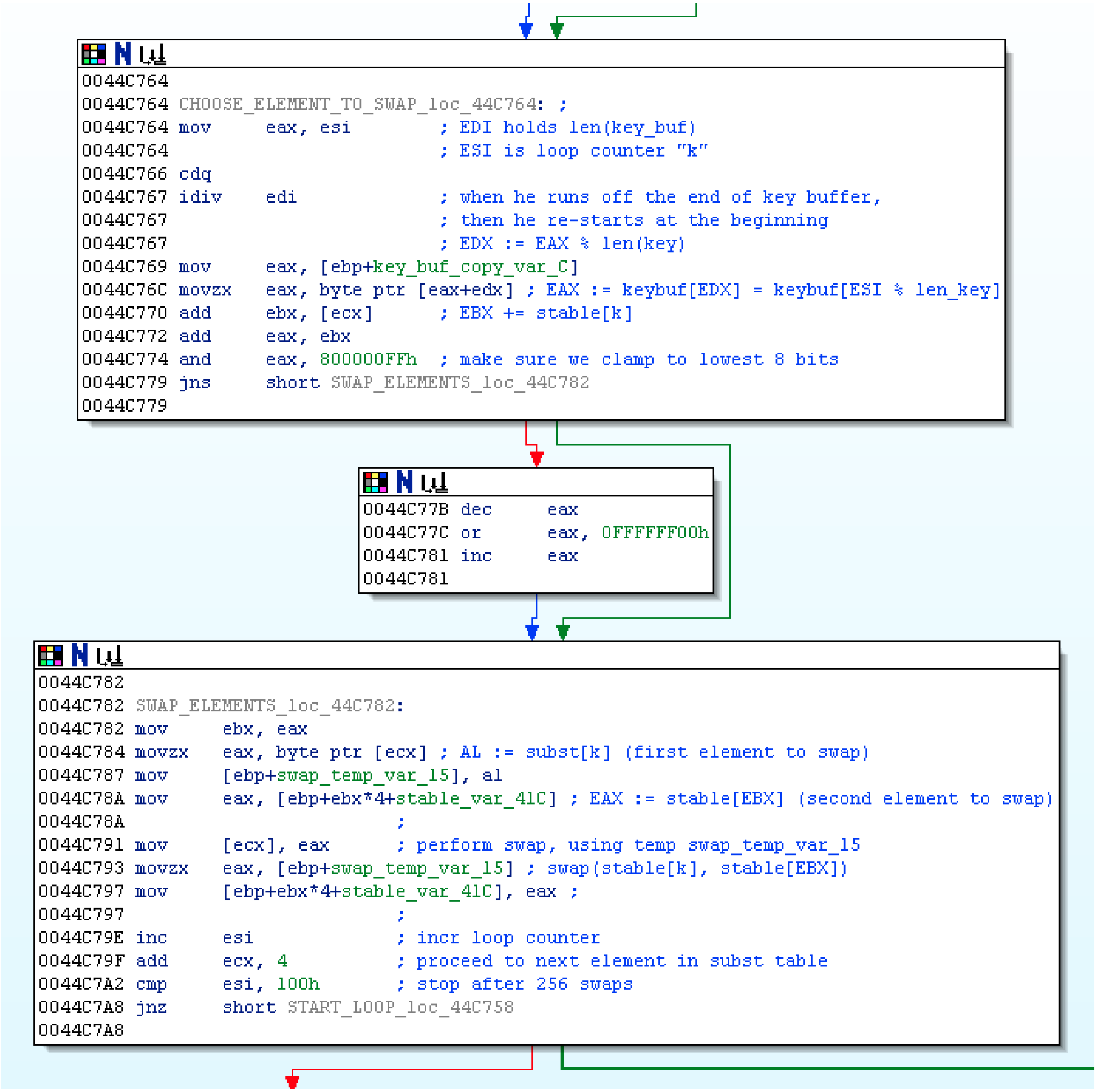
Each element in `stable_var_41C` is a 32-bit `DWORD`; the elements are initialized to the values `0x00000000` through `0x000000FF` in ascending order:

Index `ESI`	`0`	`1`	`2`	`3`	`4`	`...`	`253`	`254`	`255`
Value `subst_var_41C[ESI]`	`0x00`	`0x01`	`0x02`	`0x03`	`0x04`	`...`	`0xFD`	`0xFE`	`0xFF`

Figure 13. Initial state of substitution table `stable_var_41C`

At this point, we can guess that `stable_var_41C` is going to play the role of a substitution table for decrypting the source buffer `src_buf_var_4`, so let's see how DarkComet builds this table.

After initializing the substitution table to hold all the values between `0x00` and `0xFF` in a nice ascending order, it proceeds to vigorously scramble up the elements of the table. It makes 256 iterations through a loop; on each iteration, it swaps the positions of two of the elements in the substitution table. On the *k*th iteration, one of the swapped elements is always the *k*th element, which is pointed to by register `ECX`; the other is chosen based on the key string. The core of the loop that scrambles up the substitution table is as follows:



```
0044C764
0044C764 CHOOSE_ELEMENT_TO_SWAP_loc_44C764: ;
0044C764 mov     eax, esi        ; EDI holds len(key_buf)
0044C764                         ; ESI is loop counter "k"
0044C766 cdq
0044C767 idiv    edi             ; when he runs off the end of key buffer,
0044C767                         ; then he re-starts at the beginning
0044C767                         ; EDX := EAX % len(key)
0044C769 mov     eax, [ebp+key_buf_copy_var_C]
0044C76C movzx   eax, byte ptr [eax+edx] ; EAX := keybuf[EDX] = keybuf[ESI % len_key]
0044C770 add     ebx, [ecx]      ; EBX += stable[k]
0044C772 add     eax, ebx
0044C774 and     eax, 800000FFh  ; make sure we clamp to lowest 8 bits
0044C779 jns     short SWAP_ELEMENTS_loc_44C782
0044C779
```

```
0044C77B dec     eax
0044C77C or      eax, 0FFFFFF00h
0044C781 inc     eax
0044C781
```

```
0044C782
0044C782 SWAP_ELEMENTS_loc_44C782:
0044C782 mov     ebx, eax
0044C784 movzx   eax, byte ptr [ecx] ; AL := subst[k] (first element to swap)
0044C787 mov     [ebp+swap_temp_var_15], al
0044C78A mov     eax, [ebp+ebx*4+stable_var_41C] ; EAX := stable[EBX] (second element to swap)
0044C78A                         ;
0044C791 mov     [ecx], eax      ; perform swap, using temp swap_temp_var_15
0044C793 movzx   eax, [ebp+swap_temp_var_15] ; swap(stable[k], stable[EBX])
0044C797 mov     [ebp+ebx*4+stable_var_41C], eax ;
0044C797                         ;
0044C79E inc     esi             ; incr loop counter
0044C79F add     ecx, 4          ; proceed to next element in subst table
0044C7A2 cmp     esi, 100h       ; stop after 256 swaps
0044C7A8 jnz     short START_LOOP_loc_44C758
0044C7A8
```

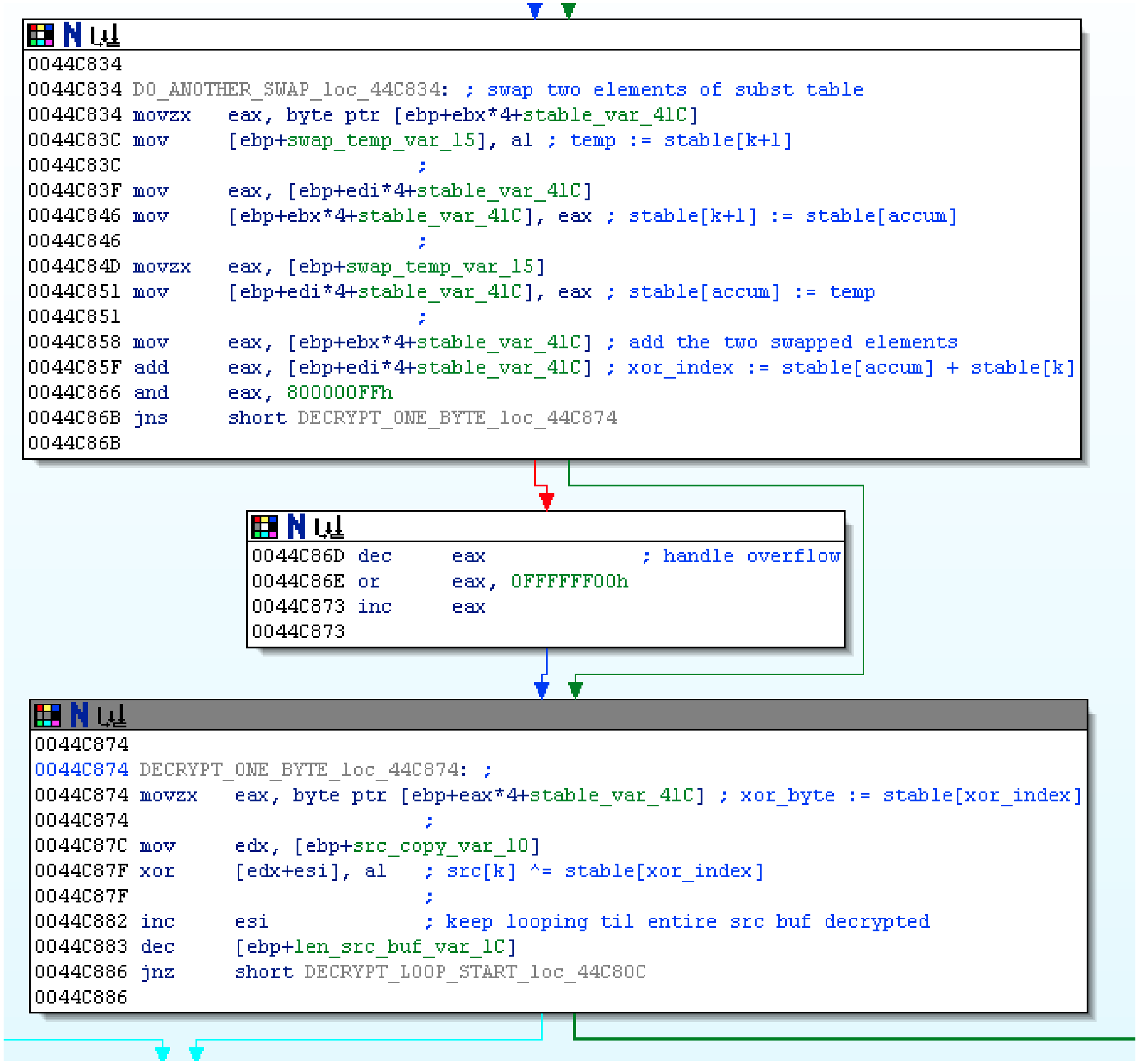
Figure 14. Function `DecryptCommandBuffer_sub_44C628()`

The first code block in the above IDA listing chooses which element of `stable_var_41C` should be swapped with the `k`th element. It uses an accumulator variable, implemented by register `EBX` and initialized to zero. On each pass through the loop, it updates the acccumulator `EBX` by adding to it the value of the `k`th element of `stable_var_41C` and the value of the current key string byte. One byte of key string is used per iteration, and whenever the key string is "used up", it restarts again at the beginning of the key; register `EDI` holds the length of the key string, so the bot just computes `k` modulo `EDI` (at instruction `0x0044C767`) to choose which byte of the key to use on the `k`th iteration.

The last code block performs the actual swapping, using `swap_temp_var_15` as the temporary variable to do the swap. Once 256 such swaps have been performed, the loop exits and the substitution table `stable_var_41C` has been nicely scrambled and is ready for use.

At this point, the actual process of decryption is performed. DarkComet iterates through its decryption loop once for each byte in the encrypted source message (after conversion from ASCII to integer representation.) The decryption loop performs the following two steps:

First, it performs an additional scrambling operation on the substitution table `stable_var_41C` by swapping two elements. When processing the `k`th source byte, the first element of the swap pair is always the `k+1`th element of table `stable_var_41C`; it uses another accumulator variable, implemented by register `EDI`, to choose the second element of the swap pair:



```
0044C834
0044C834 DO_ANOTHER_SWAP_loc_44C834: ; swap two elements of subst table
0044C834 movzx   eax, byte ptr [ebp+ebx*4+stable_var_41C]
0044C83C mov     [ebp+swap_temp_var_15], al ; temp := stable[k+1]
0044C83C                         ;
0044C83F mov     eax, [ebp+edi*4+stable_var_41C]
0044C846 mov     [ebp+ebx*4+stable_var_41C], eax ; stable[k+1] := stable[accum]
0044C846                         ;
0044C84D movzx   eax, [ebp+swap_temp_var_15]
0044C851 mov     [ebp+edi*4+stable_var_41C], eax ; stable[accum] := temp
0044C851                         ;
0044C858 mov     eax, [ebp+ebx*4+stable_var_41C] ; add the two swapped elements
0044C85F add     eax, [ebp+edi*4+stable_var_41C] ; xor_index := stable[accum] + stable[k]
0044C866 and     eax, 800000FFh
0044C86B jns     short DECRYPT_ONE_BYTE_loc_44C874
0044C86B
```

```
0044C86D dec     eax                ; handle overflow
0044C86E or      eax, 0FFFFFF00h
0044C873 inc     eax
0044C873
```

```
0044C874
0044C874 DECRYPT_ONE_BYTE_loc_44C874: ;
0044C874 movzx   eax, byte ptr [ebp+eax*4+stable_var_41C] ; xor_byte := stable[xor_index]
0044C874                         ;
0044C87C mov     edx, [ebp+src_copy_var_10]
0044C87F xor     [edx+esi], al    ; src[k] ^= stable[xor_index]
0044C87F                         ;
0044C882 inc     esi              ; keep looping til entire src buf decrypted
0044C883 dec     [ebp+len_src_buf_var_1C]
0044C886 jnz     short DECRYPT_LOOP_START_loc_44C80C
0044C886
```

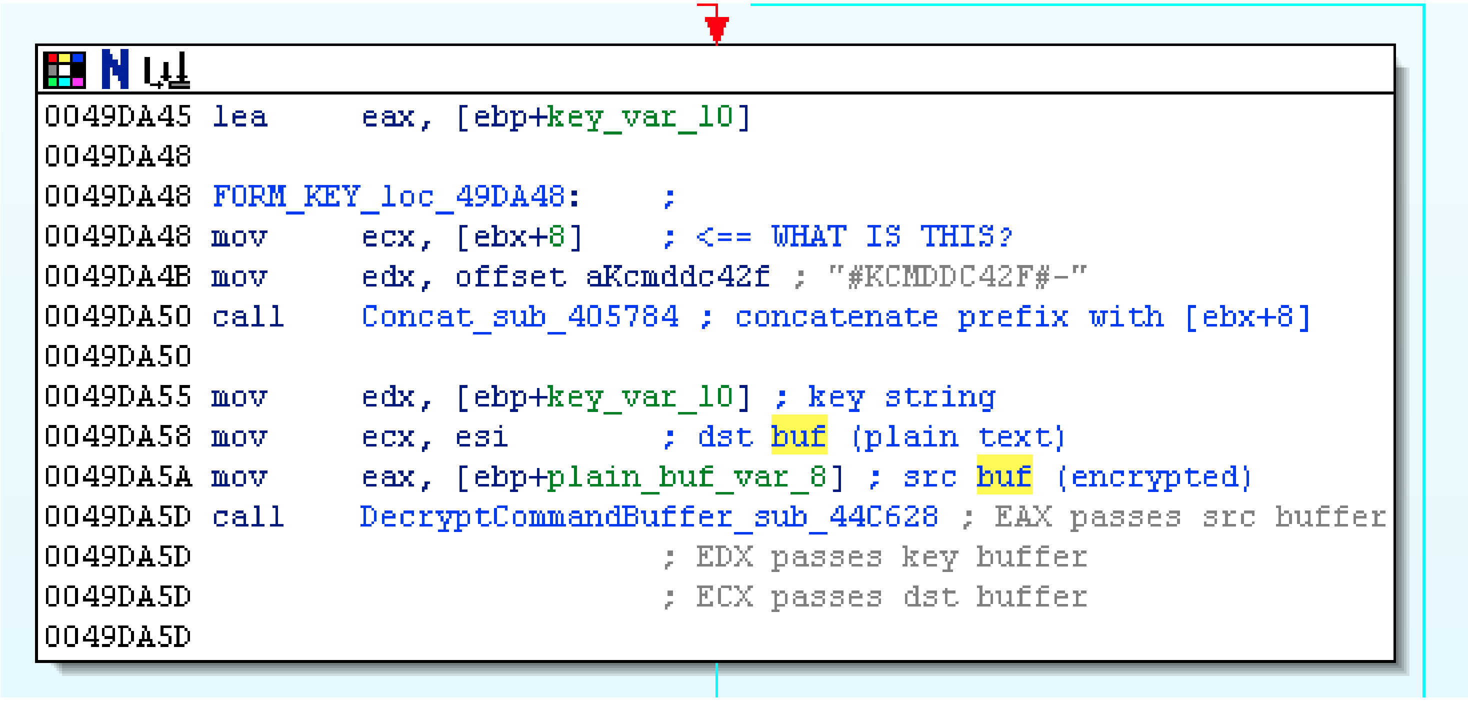
Figure 15. Function `DecryptCommandBuffer_sub_44C628()`

After performing this swap operation, DarkComet finally decrypts a byte of message. It sums up the values of the two swapped elements (at instruction `0x0044C85F`), then uses the result (modulo 256) to re-index into the `stable_var_41C` table to pull out a third element (at instruction `0x0044C874`). This third element is `XOR`ed against the current (`k`th) source byte to produced a decrypted character.

It should be pointed out that conceptually, this decryption mechanism - both the manner in which the substitution table is built, as well as how it is used for `XOR`-based decryption - is *very* similar to that used by the Trojan.PonyDOS malware family. The actual implementation has quite a few differences, but the basic encryption algorithm is the same. Trojan.PonyDOS, however, adds a few additional layers to secure its communications protocol above and beyond the core crypto algorithm which it shares with DarkComet; specifically, the computation of some cryptographic hashes. Also, Trojan.PonyDOS does not go to the trouble of converting its encrypted data payloads into ASCII representations as DarkComet does.

Now that we've reversed the core DarkComet decryption mechanism (needed to read C&C commands), we'll want to confirm that the encryption mechanism (needed to read and/or fake bot phone home messages) is symmetric. And indeed, by following references to the socket handle used to `recv()` the initial C&C command, we can trace through to find the encryption routine called by DarkComet just prior to `send()`ing back its response messages. Sure enough, the encryption routine, `Encrypt_sub_44C34C()`, is functionally identical to the decryption routine, as hoped and expected; the only difference being that the `Integerize_sub_44C1C0()` routine prior to decryption is absent, and a new routine, which we'll call `Integer2String_sub_409C6C()`, is called following the encryption step; this routine simply converts the raw encrypted data back into the ASCII version of its hexadecimal values.

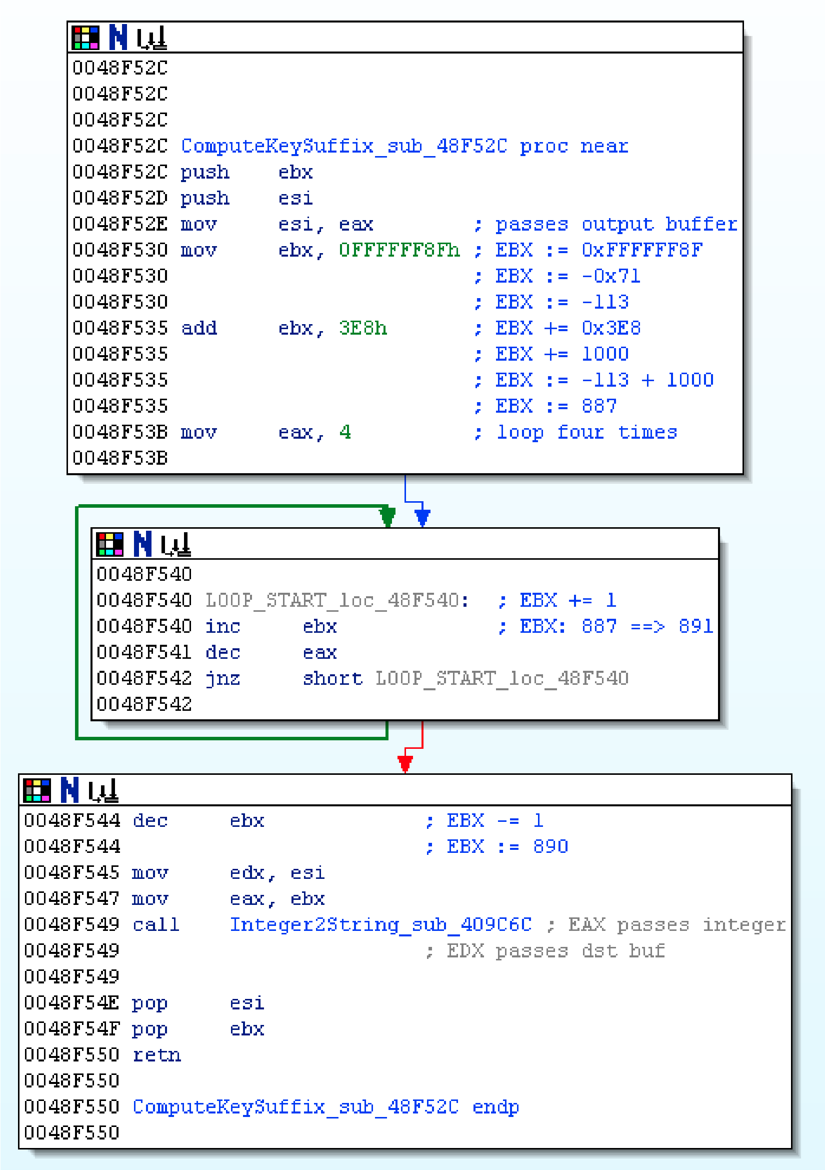
Of course, in order to have a fully functional implementation of DarkComet's crypto system, we'll need to know what key strings it uses. We see that there are two locations where `DecryptCommandBuffer_sub_44C628()` is called, and one of those locations, `EncryptData_sub_49D9EC()`, has a hard-coded string with an uncanny resemblance to a decryption key:



```
0049DA45 lea     eax, [ebp+key_var_10]
0049DA48
0049DA48 FORM_KEY_loc_49DA48:    ;
0049DA48 mov     ecx, [ebx+8]    ; <== WHAT IS THIS?
0049DA4B mov     edx, offset aKcmddc42f ; "#KCMDDC42F#-"
0049DA50 call    Concat_sub_405784 ; concatenate prefix with [ebx+8]
0049DA50
0049DA55 mov     edx, [ebp+key_var_10] ; key string
0049DA58 mov     ecx, esi        ; dst buf (plain text)
0049DA5A mov     eax, [ebp+plain_buf_var_8] ; src buf (encrypted)
0049DA5D call    DecryptCommandBuffer_sub_44C628 ; EAX passes src buffer
0049DA5D                         ; EDX passes key buffer
0049DA5D                         ; ECX passes dst buffer
0049DA5D
```

Figure 16. Function `EncryptData_sub_49D9EC()`

We see that the decryption string `key_var_10`, passed to `DecryptCommandBuffer_sub_44C628()` via `EDX`, is formed by concatenating a hard-coded string `#KCMDDC42F#-` with some mystery string stored at `[EBX+8]`. It turns out that this mysterious value stored at an offset from `EBX` is passed into `EncryptData_sub_49D9EC()` via the `EAX` register. Tracing backwards up the stack, we follow the reference to `EAX` as the baton is passed from register to register. It does not take long to come across the following routine, which we will label `ComputeKeySuffix_sub_48F52C()`:



```
0048F52C
0048F52C
0048F52C
0048F52C ComputeKeySuffix_sub_48F52C proc near
0048F52C push    ebx
0048F52D push    esi
0048F52E mov     esi, eax         ; passes output buffer
0048F530 mov     ebx, 0FFFFFF8Fh ; EBX := 0xFFFFFF8F
0048F530                         ; EBX := -0x71
0048F530                         ; EBX := -113
0048F535 add     ebx, 3E8h       ; EBX += 0x3E8
0048F535                         ; EBX += 1000
0048F535                         ; EBX := -113 + 1000
0048F535                         ; EBX := 887
0048F53B mov     eax, 4          ; loop four times
0048F53B
```

```
0048F540
0048F540 LOOP_START_loc_48F540:  ; EBX += 1
0048F540 inc     ebx             ; EBX: 887 ==> 891
0048F541 dec     eax
0048F542 jnz     short LOOP_START_loc_48F540
0048F542
```

```
0048F544 dec     ebx             ; EBX -= 1
0048F544                         ; EBX := 890
0048F545 mov     edx, esi
0048F547 mov     eax, ebx
0048F549 call    Integer2String_sub_409C6C ; EAX passes integer
0048F549                         ; EDX passes dst buf
0048F549
0048F54E pop     esi
0048F54F pop     ebx
0048F550 retn
0048F550
0048F550 ComputeKeySuffix_sub_48F52C endp
0048F550
```

Figure 17. Function `ComputeKeySuffix_sub_48F52C()`

You don't run into code like this very often. It receives an output buffer passed via `EAX`. It then uses register `EBX` to do some rather "inefficient" operations. First, it assigns `EBX` the value `0xFFFFFF8F`, or -71. It then adds 1000 to `EBX`, yielding 887. Then it goes through four iterations of a loop that has no purpose other than to increment `EBX` by one on each iteration, resulting in a value of 891. Finally, it completes its laborious calculations by decrementing EBX by one, yielding a final answer of 890. This integer is passed to a standard integer-to-string API, which writes the string `890` into the output buffer. In C, these shenanigans would look something like the following:

```
int nAddend = 1000;
int nSuffix = -71;
int nResult = nSuffix + nAddend;
for (int k=0; k<4; k++)
    nResult += 1;
sprintf(suffix, "%d", --nResult);
```

This is a very roundabout way of assigning the hard-coded string `890` to a buffer. Clearly the DarkComet author is (wisely) trying to avoid having the entire decryption key string hard-coded in the bot executable.

So at this point, we know that the decryption key is composed of the prefix `#KCMDDC42F#-` concatenated with the suffix `890`, yielding `#KCMDDC42F#-890`.

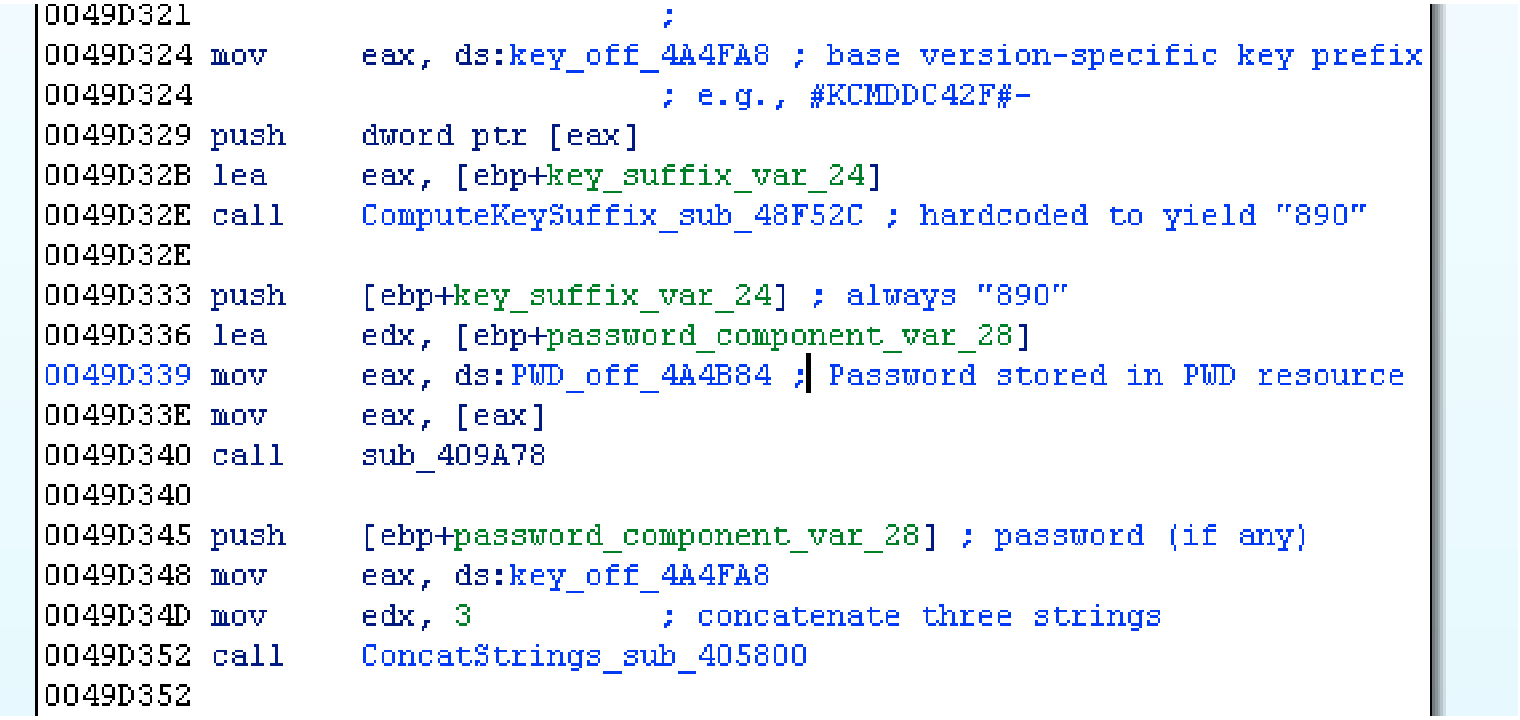
One final note regarding the encryption key strings used by DarkComet: as first documented in Contextis' Laura Aylward's DarkComet analysis, each version of DarkComet uses a different hard-coded string for the key prefix. For example, we have observed the following:

Dark Comet version	**Crypto Key Prefix (Default)**
Version 4.0	`#KCMDDC4#-890`
Version 4.2	`#KCMDDC42F#-890`

Version 5.0	#KCMDDC5#-890

Figure 18. Standard crypto key prefixes for DarkComet versions

Furthermore, and also documented by Contextis, DarkComet supports the use of an optional password that is appended to the default (version-specific) crypto key. For example, the default password (if enabled) string is `0123456789`. This 10-digit string will be appended to the standard crypto key `#KCMDDC42F#-890` (in the case of DarkComet version 4.2) to yield a final key of `#KCMDDC42F#-8900123456789`. The code that performs this concatenation is found in a routine we'll call `FormCryptoKey_sub_49D2F4()`:



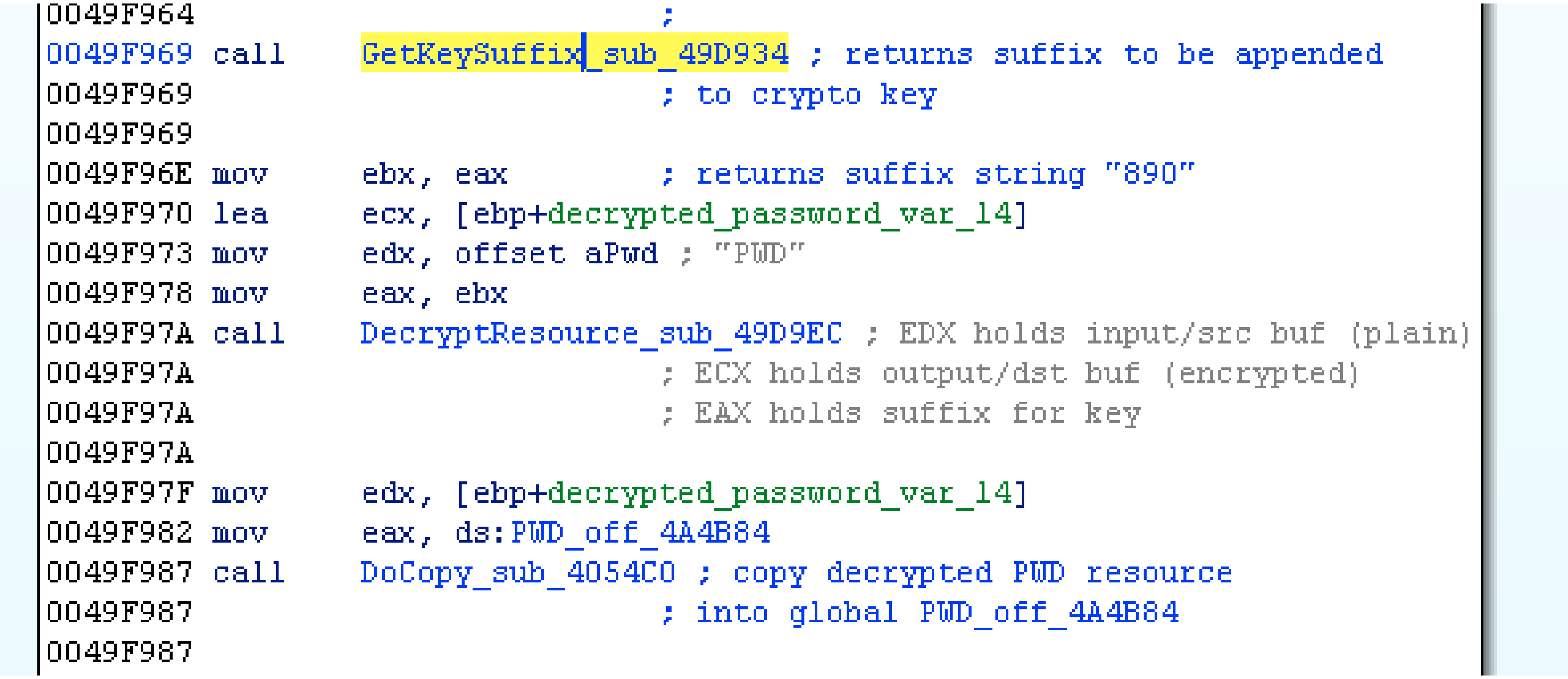
```
0049D321                         ;
0049D324 mov     eax, ds:key_off_4A4FA8 ; base version-specific key prefix
0049D324                         ; e.g., #KCMDDC42F#-
0049D329 push    dword ptr [eax]
0049D32B lea     eax, [ebp+key_suffix_var_24]
0049D32E call    ComputeKeySuffix_sub_48F52C ; hardcoded to yield "890"
0049D32E
0049D333 push    [ebp+key_suffix_var_24] ; always "890"
0049D336 lea     edx, [ebp+password_component_var_28]
0049D339 mov     eax, ds:PWD_off_4A4B84 ; Password stored in PWD resource
0049D33E mov     eax, [eax]
0049D340 call    sub_409A78
0049D340
0049D345 push    [ebp+password_component_var_28] ; password (if any)
0049D348 mov     eax, ds:key_off_4A4FA8
0049D34D mov     edx, 3          ; concatenate three strings
0049D352 call    ConcatStrings_sub_405800
0049D352
```

Figure 19. Function `FormCryptoKey_sub_49D2F4()`

This code concatenates the three components of the final crypto key: the hard-coded prefix (e.g., `#KCMDDC42F#-`), the three-digit string `890` that is not technically hard-coded but deterministically computed using the aforementioned `ComputeKeySuffix_sub_48F52C()` routine, and the optional botnet password stored in the global variable `PWD_off_4A4B84`.

The password itself is actually stored as an encrypted resource. Upon initialization, it is decrypted using a preliminary crypto key comprised only of the first two components (e.g., `#KCMDDC42F#-890`) using a routine we've labeled `DecryptResource_sub_49D9EC()`. To make a long story short, this routine uses the Windows APIs `FindResource()`, `LoadResource()`, etc. to extract a named resource of type `RT_RCDATA` (code `0x0A`), intended for "application-defined resources (raw data)". The raw data is then decrypted using the preliminary crypto key.

In the case of the crypto password, the name of the resource is `PWD`. The resource is extracted, decrypted, and stored for future use in the global variable `PWD_off_4A4B84` by a function we call `DecryptResources_sub_49F92C()`:



```
0049F964                         ;
0049F969 call    GetKeySuffix_sub_49D934 ; returns suffix to be appended
0049F969                         ; to crypto key
0049F969
0049F96E mov     ebx, eax        ; returns suffix string "890"
0049F970 lea     ecx, [ebp+decrypted_password_var_14]
0049F973 mov     edx, offset aPwd ; "PWD"
0049F978 mov     eax, ebx
0049F97A call    DecryptResource_sub_49D9EC ; EDX holds input/src buf (plain)
0049F97A                         ; ECX holds output/dst buf (encrypted)
0049F97A                         ; EAX holds suffix for key
0049F97A
0049F97F mov     edx, [ebp+decrypted_password_var_14]
0049F982 mov     eax, ds:PWD_off_4A4B84
0049F987 call    DoCopy_sub_4054C0 ; copy decrypted PWD resource
0049F987                         ; into global PWD_off_4A4B84
0049F987
```

Figure 20. Function `DecryptResources_sub_49F92C()`

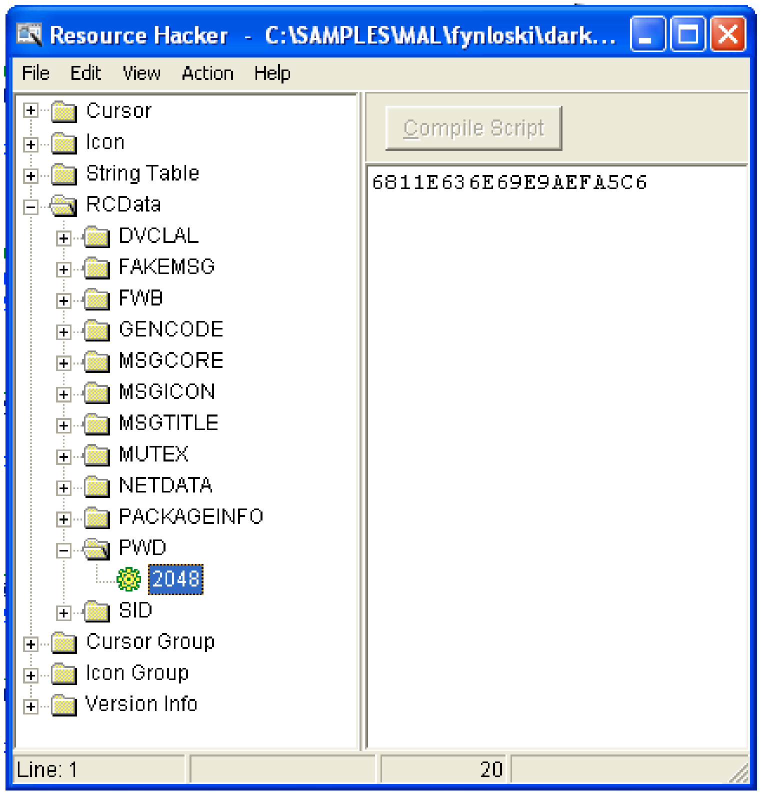
In the case of the default password `0123456789`, the encrypted resource will hold the value `6811E636E69E9AEFA5C6`. This `DecryptResources_sub_49F92C()` function actually decrypts a lot of encrypted bot parameters stored in various resources; some of the more interesting ones are as follows:

Resource Name	Encrypted Data	Decrypted Value
`FAKEMSG`	`69`	`1`
`GENCODE`	`6146B749A3CF9C9FE8CFAB2C`	`9fcLqd0Gu00j`
`MSGCORE`	`1100A768B3C7C0F8FCDFC907B6F9`	`I small a RAT!`
`MSGTITLE`	`1C41A66E91C4C1BDE9`	`DarkComet`

MUTEX	1C638B4887FFE980B0B9AE72B1EA40A3	DC_MUTEX-F54S21D
NETDATA	6919E62BE39D94F6ACCFAB68D5ED4BD67BA333	192.168.100.75:1604
PWD	6811E636E69E9AEFA5C6	0123456789
SID	1F55B176A69A9A	Guest16

Figure 21. Interesting encrypted resources

Of particular interest is the encrypted `NETDATA` resource, which holds the C&C hostname and port. The Resource Hacker tool is a great utility for viewing and extracting the various DarkComet encrypted parameters:



```
6811E636E69E9AEFA5C6
```

Figure 22. Resource Hacker extracting DarkComet resources

So to summarize, DarkComet uses a hard-coded (although different for each version) preliminary key string, such as `#KCMDDC42F#-890`, to decrypt its sensitive parameters from various raw resources - such as the C&C information and communications password stored in the `NETDATA` and `PWD` resources, respectively. It then appends the decrypted comms password (stored in the `PWD` resource) to the end of the preliminary crypto key string to form the final key, `#KCMDDC42F#-8900123456789`, that it uses for securing the network traffic to and from its C&C server.

Putting everything together into a complete DarkComet crypto module yields the following Python script:

```
# DarkComet decryptor/encryptor
# Copyright (c) 2012 Arbor Networks

import sys

class DarkCometCryptor(object):

    def __init__(self, key):
        self._len_key = len(key)
        self._key = [ord(token) for token in key]

    def decrypt(self, src):
        # Convert ASCII to hex representation
        buf = [int("0x%s" % src[k*2:k*2+2], 16) for k in range(len(src)//2)]
        self._cryption(buf)
        return "".join([chr(token) for token in buf])

    def encrypt(self, src):
        buf = [ord(token) for token in src]
```

```
        self._cryption(buf)
        # Convert to hex codes (upper case)
        return "".join(["%02x" % tok for tok in buf]).upper()

    def _cryption(self, src):
        # Build subst table
        stable = list(range(256))
        accum = 0
        for k in range(256):
            accum += stable[k]
            accum += self._key[k % self._len_key]
            accum &= 0xff
            stable[k], stable[accum] = stable[accum], stable[k]
        # Apply subst table
        accum = 0
        for k in range(len(src)):
            elem_a_idx = self._LS_BYTE(k + 1)
            accum += stable[elem_a_idx]
            elem_b_idx = self._LS_BYTE(accum)
            stable[elem_b_idx], stable[elem_a_idx] = \
                       stable[elem_a_idx], stable[elem_b_idx]
            swap_sum = self._LS_BYTE(stable[elem_b_idx] + stable[elem_a_idx])
            src[k] ^= self._LS_BYTE(stable[swap_sum])

    @staticmethod
    def _LS_BYTE(value):
        return 0xff & value


if __name__ == '__main__':
    if len(sys.argv) != 4 or sys.argv[1] not in ('-d', '-e'):
        print "usage: %s [-d|-e] SRC_TEXT KEY" % sys.argv[0]
```

```
        sys.exit(1)
    do_decrypt = bool(sys.argv[1] == '-d')
    src = sys.argv[2]
    key = sys.argv[3]
    print "%s: %s" % ("CRYPT" if do_decrypt else "PLAIN", src)
    cryptor = DarkCometCryptor(key)
    dst = cryptor.decrypt(src) if do_decrypt else cryptor.encrypt(src)
    print "%s: %s" % ("PLAIN" if do_decrypt else "CRYPT", dst)
```

Figure 23. `darkcomet.py` Crypto Module

Applying our DarkComet encryption module against the observed traffic results in the following:

```
C&C:
IDTYPE
Bot:
SERVER
C&C:
GetSIN192.10.8.64|27038511
Bot:
infoesComet|192.10.8.64 / [192.1.167.30] : 1604|SANDBOX7 /
Admin|27038511|29s|Windows XP Service Pack 2 [2600] 32 bit ( C:\
)|x||US|C:\WINDOWS\system32\cmd.exe|{16382783-b70c-71e4-11e0-
28f8efc0696f-10806d6172}|127.43 MiB/256.09 MiB [128.22 MiB Free]|English
(United States) US /  -- |10/9/2011 at 8:13:31 PM
```

Figure 24. Decrypted version of comms from Figure 2.

Likewise, when a DarkComet C&C issues attacks command, the encrypted traffic on the wire looks like these examples:

```
185CB63BBE0EA3DF6D2A725936265160E391BC77F47FF46A3934CFB173AC
```

```
185CB63BA31EA7C967297252432E5A7CFC96B261EB7EF4742533CEBF37A9C081
185CB63BA503B9C967297252432E5A7CFC96B261EB7EF4742533CEBF37A9C081
```

But applying the decryption routine yields the following:

```
DDOSHTTPFLOOD192.168.100.254|5
DDOSUDPFLOOD192.168.100.254:80|5
DDOSSYNFLOOD192.168.100.254:80|5
```

Which corresponds to ordering an HTTP flood, a UDP flood, and a TCP flood, respectively, against target `192.168.100.254`, with each attack lasting for 5 seconds. Once the attacks are completed the DarkComet bot will respond with an encrypted status message such as the following:

```
1E4CAB2DA50FBBDB781F5336347B073DA9DCD936B46EB03B646DDAE366F7D5C76D3C0420A55906F524
240A0F34D3A6384150
```

Which decrypts to the following:

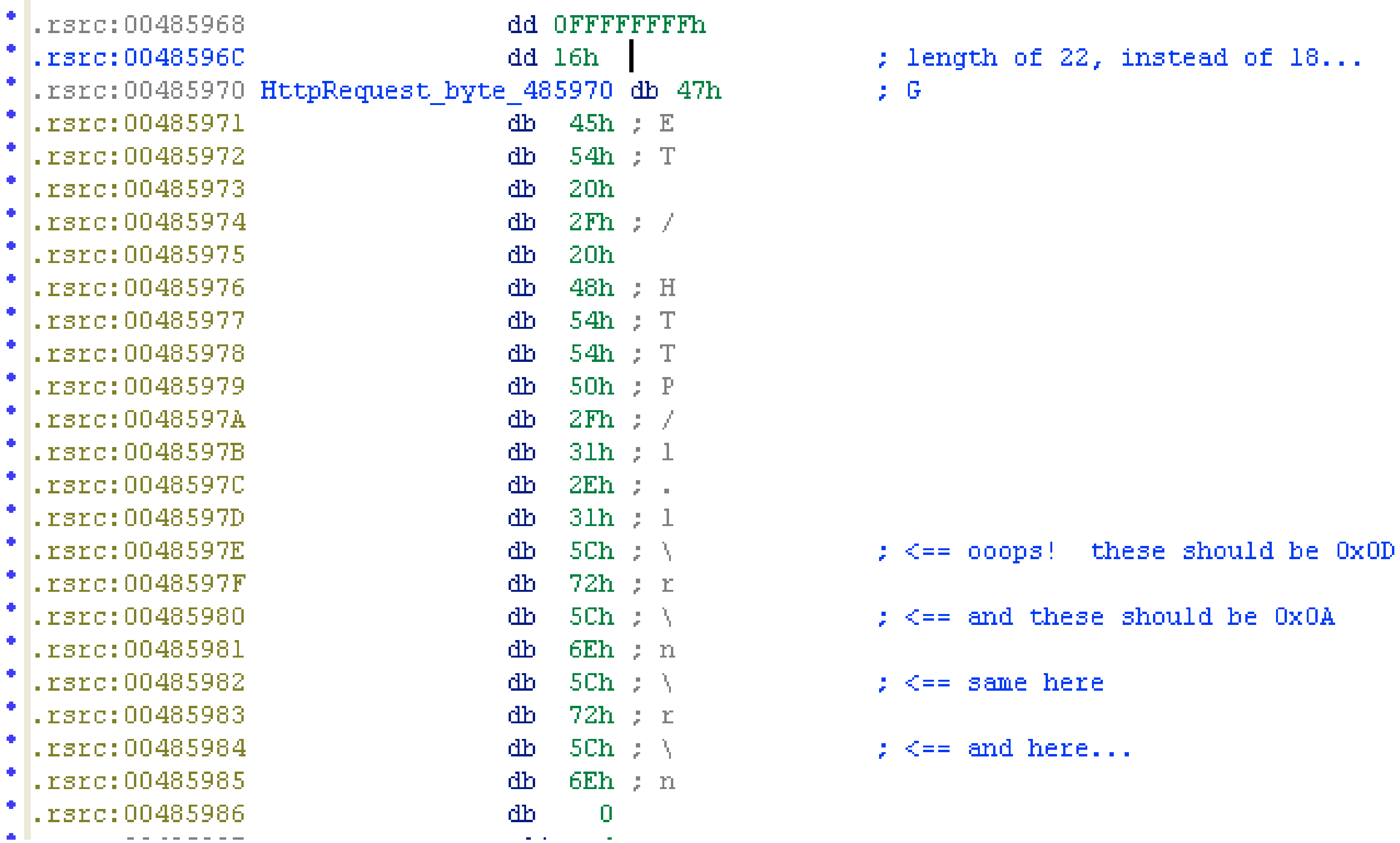
```
BTRESULTSyn Flood|Syn task finished!|Administrator
```

As implied above, DarkComet supports three types of DDoS attacks: HTTP flooding, UDP flooding, and TCP flooding (mis-advertised as "`SYNFLOOD`"). The UDP and TCP volumetric floods are quite unremarkable and simply consist of random gibberish blasted at a target host and port. The HTTP flood also appears to be intended as a rudimentary GET flood with a minimalist HTTP request header. However, DarkComet's HTTP flood implementation happens to have not one, but two catastrophic bugs.

First of all, the thread procedure that implements the `DDOSHTTPFLOOD` attack command, `SendHttp_sub_485848()`, uses the WinSock2 library's `socket()`, `connect()`, and `send()` APIs to send the following hard-coded HTTP flooding request:

```
GET / HTTP/1.1\r\n\r\n
```

At first glance, this looks like an (almost) valid, although minimalist, HTTP request that is terminated with a double carriage-return/line-feed (`CRLF`) combination. However, when one takes a closer look at the way DarkComet stores this string, we see that the `\r` and `\n` characters are not actually `CR` (`0x0D`) and `LF` (`0x0A`) bytes. Instead, they are literally comprised of the backslash (`0x2F`), letter `r` (`0x72`), and letter `n` (`0x6E`) bytes!

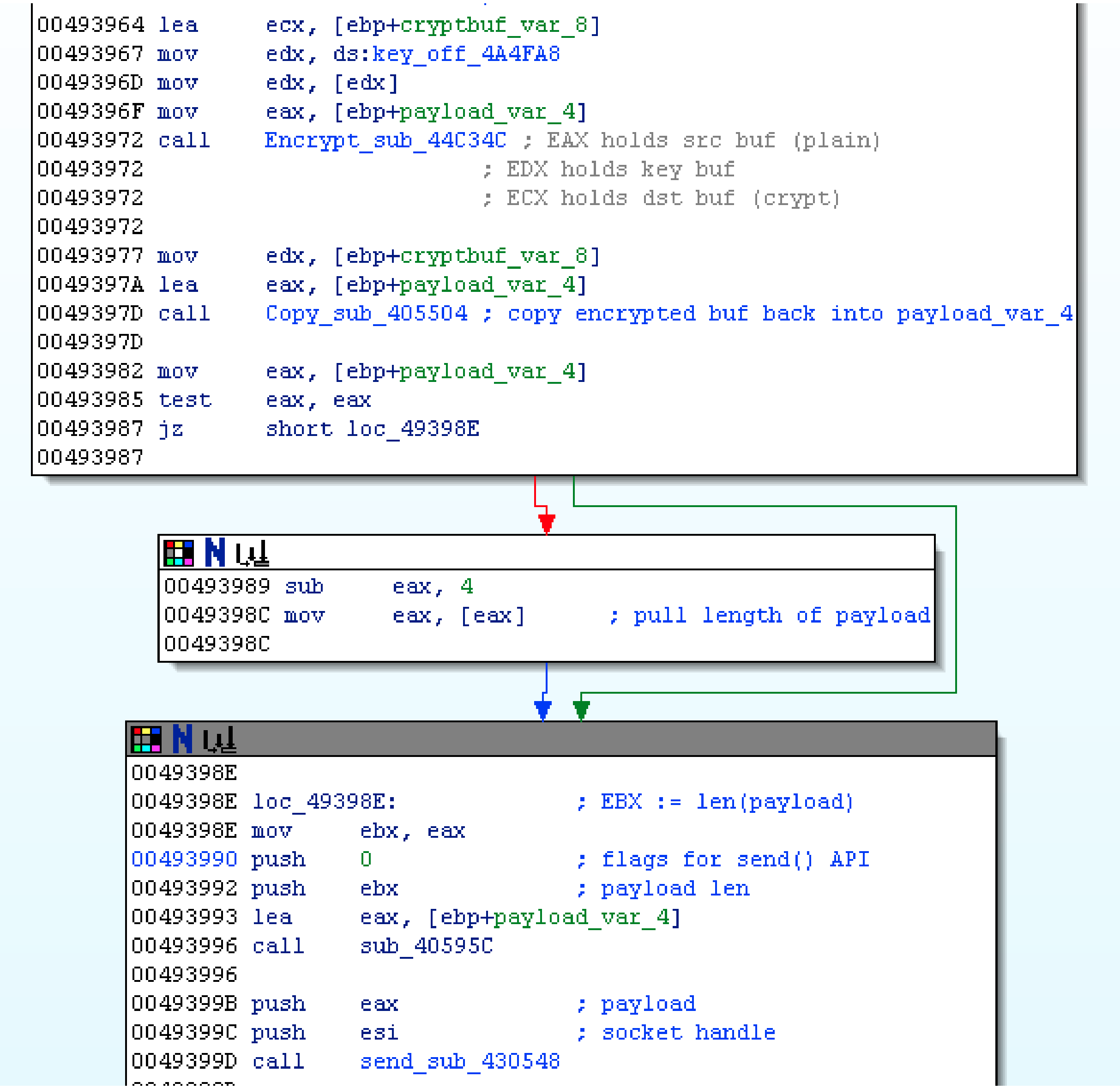


```
.rsrc:00485968                  dd 0FFFFFFFFh
.rsrc:0048596C                  dd 16h                      ; length of 22, instead of 18...
.rsrc:00485970 HttpRequest_byte_485970 db 47h               ; G
.rsrc:00485971                  db  45h ; E
.rsrc:00485972                  db  54h ; T
.rsrc:00485973                  db  20h
.rsrc:00485974                  db  2Fh ; /
.rsrc:00485975                  db  20h
.rsrc:00485976                  db  48h ; H
.rsrc:00485977                  db  54h ; T
.rsrc:00485978                  db  54h ; T
.rsrc:00485979                  db  50h ; P
.rsrc:0048597A                  db  2Fh ; /
.rsrc:0048597B                  db  31h ; 1
.rsrc:0048597C                  db  2Eh ; .
.rsrc:0048597D                  db  31h ; 1
.rsrc:0048597E                  db  5Ch ; \                 ; <== ooops!  these should be 0x0D
.rsrc:0048597F                  db  72h ; r
.rsrc:00485980                  db  5Ch ; \                 ; <== and these should be 0x0A
.rsrc:00485981                  db  6Eh ; n
.rsrc:00485982                  db  5Ch ; \                 ; <== same here
.rsrc:00485983                  db  72h ; r
.rsrc:00485984                  db  5Ch ; \                 ; <== and here...
.rsrc:00485985                  db  6Eh ; n
.rsrc:00485986                  db    0
```

Figure 25. Hard-coded HTTP request string `HttpRequest_byte_485970`

If the HTTP request string had been encoded properly (ending with `0x0D0A0D0A`), the length of the string would have been 18. But instead, we see that it is 22 bytes in length. Due to this, DarkComet's attempt at an application layer attack is not close to a valid HTTP request per the RFCs.

The second big mistake in the implementation of DarkComet's HTTP flood attack becomes apparent further down in the attack thread code, just before the (buggy) HTTP request payload is sent to the target via the `send()` API:



```
00493964 lea     ecx, [ebp+cryptbuf_var_8]
00493967 mov     edx, ds:key_off_4A4FA8
0049396D mov     edx, [edx]
0049396F mov     eax, [ebp+payload_var_4]
00493972 call    Encrypt_sub_44C34C ; EAX holds src buf (plain)
00493972                           ; EDX holds key buf
00493972                           ; ECX holds dst buf (crypt)
00493972
00493977 mov     edx, [ebp+cryptbuf_var_8]
0049397A lea     eax, [ebp+payload_var_4]
0049397D call    Copy_sub_405504 ; copy encrypted buf back into payload_var_4
0049397D
00493982 mov     eax, [ebp+payload_var_4]
00493985 test    eax, eax
00493987 jz      short loc_49398E
00493987
```

```
00493989 sub     eax, 4
0049398C mov     eax, [eax]      ; pull length of payload
0049398C
```

```
0049398E
0049398E loc_49398E:             ; EBX := len(payload)
0049398E mov     ebx, eax
00493990 push    0               ; flags for send() API
00493992 push    ebx             ; payload len
00493993 lea     eax, [ebp+payload_var_4]
00493996 call    sub_40595C
00493996
0049399B push    eax             ; payload
0049399C push    esi             ; socket handle
0049399D call    send_sub_430548
```

Figure 26. Function `EncryptAndSendData_sub_49393C()`

Unbelievably, DarkComet bot is accidentally encrypting the (buggy) `GET` request string at instruction `0x00493972` via a call to the already-reversed `Encrypt_sub_44C34C()` routine. The resulting (encrypted) HTTP request is then sent on its merry way to the DDoS target via the `send()` API call at instruction `0x0049399D`.

So the target web server ends up receiving gibberish instead of a well-formed HTTP request that might exhaust resources at the application layer. Due to these two serious flaws, DarkComet's HTTP flood attack reduces down to nothing more than a volumetric TCP flood against port 80, and a very weak one at that (a mere 22 bytes of TCP payload per flooding packet...) In fact, here is what the actual "HTTP flooding" traffic looks like:

```
1B5DAD48D97ABFDB7F3612275C26342091CED63D8620
1B5DAD48D97ABFDB7F3612275C26342091CED63D8620
1B5DAD48D97ABFDB7F3612275C26342091CED63D8620
```

Clearly, this is very unlikely to bring any web server to its knees!

Acknowledgements to Arbor Networks analyst Curt Wilson for his valuable insights and assistance with this article.