It's 2012 and Armageddon has arrived

Breaking Armageddon's latest and greatest crypto reveals some interesting new functionality - 3/6/2012

Jeff Edwards, Research Analyst, Arbor Networks ASERT

Armageddon is one of several notable Russian malware families that are designed exclusively for DDoS attacks; it has been on our radar screens for some time now. Its primary competitors within the market of Russian DDoS vendors are Dirt Jumper (a.k.a. RussKill), Darkness/Optima (a.k.a. Votwup), and ofcourse BlackEnergy.

We've noticed that the Armageddon code base has undergone some relatively rapid evolution lately, and the purpose of this blog post is to report on some of the new functionality we have observed. With this latest release, the bot uses some new crypto protection to hide its features from casual observers; breaking this encryption revealed some interesting goodies...

This paper is also intended to be the first in an upcoming series of articles that will provide a guided tour of the inner workings of various crypto systems that are used by contemporary DDoS malware families in order to hide their communications and sensitive data - and how to go about breaking them.

Breaking Armageddon's Crypto

We use IDA Pro as our primary reversing tool, but since Armageddon is written in Delphi, we supplement IDA with the excellent Interactive Delphi Reconstructor (IDR) tool. IDR often does a pretty good job at automatically detecting many/most of the classes, properties, and methods included in the various versions of the Delphi application framework. It also tends to work well for common third-party libraries such as the Synapse package that is often used by Delphi-based malware to perform network operations, such as phoning home to command & control (C&C) servers, generating DDoS flood traffic, etc. One can often minimize the amount of time wasted digging through layer upon layer of Delphi function calls by simultaneously opening the malware in both IDA and IDR, and using IDR as a reference to identify Delphi functions, while doing the bulk of the analysis in IDA.

The sample bot that we will analyze in this post is 45,056 bytes in size with MD5 hash of `5ae3451b76c76c20e6fe58289bb8302c`. As usual, a great place to start the reversing of a bot is to run a `strings` dump on the executable and identify any interesting clues therein. Applying this approach to our Armageddon sample does not yield much at first glance, except for about 45 strings that appear to be encrypted in some fashion; here are some examples:

```
70D997203F8AD33CE175FA6CF7D5396A0739D0E8487BCE4568B40BAC92A1BE
A405ED14624D0D3258D403EC8ABD56B548961379C7
9B7AEB04381B284D43CF45B5
```

Earlier versions of Armageddon bots stored most of their sensitive strings in plain ASCII; this encryption is a new development. So it seems we must first reverse this crypto before the inner workings of the bot are revealed.

The encrypted strings pretty much stand out like a sore thumb, as they all consist of sequences of ASCII characters drawn from the set of digits and the upper-case letters `A` through `F`; furthermore, the length of these sequences is always an even number. Clearly, these strings represent sequences of hexadecimal numbers.

So we'll open the bot in IDA and locate where one of the encrypted strings (`70D997203F8AD33CE`...) is referenced:

IDA View-A

```
0040A035 lea     edx, [ebp+var_C]
0040A038 mov     eax, 184B30h
0040A03D call    sub_403694
0040A03D
0040A042 mov     eax, [ebp+var_C]
0040A045 push    eax
0040A046 lea     edx, [ebp+var_10]
0040A049 mov     eax, offset a70d997203f8ad3 ; "70D997203F8AD33CE175FA6CF7D5396A0739D0E"...
0040A04E call    sub_405514
0040A04E
0040A053 mov     eax, [ebp+var_10]
0040A056 lea     ecx, [ebp+var_8]
0040A059 pop     edx
0040A05A call    sub_405714
0040A05A
0040A05F mov     edx, [ebp+var_8]
0040A062 mov     eax, offset dword_40C69C
0040A067 call    sub_401EE4
0040A067
0040A06C lea     edx, [ebp+var_18]
0040A06F mov     eax, 184B30h
0040A074 call    sub_403694
0040A074
0040A079 mov     eax, [ebp+var_18]
0040A07C push    eax
0040A07D lea     edx, [ebp+var_1C]
0040A080 mov     eax, offset a70d997203f8a_0 ; "70D997203F8AD32A83149F1DEFD5396A0739D3E"...
0040A085 call    sub_405514
0040A085
0040A08A mov     eax, [ebp+var_1C]
0040A08D lea     ecx, [ebp+var_14]
0040A090 pop     edx
0040A091 call    sub_405714
```

100.00% | (4,1653) | (529,439) | 00009467 | 0040A067: sub_409F30+137

Function `sub_409F30()`

Hmmm, we see that our encrypted string of interest is passed as the first argument (via `EAX`, as is the convention with Delphi) to a certain `sub_405514()`. We also notice that another (though similar) encrypted string is passed to the same function a little later. So this `sub_405514()` seems like a good place to start looking for a decryption routine.
In addition to the encrypted string being passed in via register `EAX`, and it looks like perhaps a destination buffer (local stack variable `var_10`) is passed through `EDX`. Taking a look at the graph of `sub_405514()`, we see that it basically operates in a single loop, and that the main code block within this loop makes a number of function calls:

IDA View-A

```
00405563 mov     edx, offset dword_40560C
00405568 lea     eax, [ebp+var_10]
0040556B call    sub_40146C
00405570 lea     eax, [ebp+var_14]
00405573 mov     edx, [ebp+var_4]
00405576 mov     dl, [edx+ebx-1]
0040557A mov     [eax+1], dl
0040557D mov     byte ptr [eax], 1
00405580 lea     edx, [ebp+var_14]
00405583 lea     eax, [ebp+var_10]
00405586 mov     cl, 2
00405588 call    sub_40143C
0040558D lea     edx, [ebp+var_10]
00405590 lea     eax, [ebp+var_18]
00405593 call    sub_40146C
00405598 lea     eax, [ebp+var_14]
0040559B mov     edx, [ebp+var_4]
0040559E mov     dl, [edx+ebx]
004055A1 mov     [eax+1], dl
004055A4 mov     byte ptr [eax], 1
004055A7 lea     edx, [ebp+var_14]
004055AA lea     eax, [ebp+var_18]
004055AD mov     cl, 3
004055AF call    sub_40143C
004055B4 lea     edx, [ebp+var_18]
004055B7 lea     eax, [ebp+var_C]
004055BA call    sub_401FEC
004055BF mov     eax, [ebp+var_C]
004055C2 call    sub_403688
004055C7 mov     edx, eax
004055C9 lea     eax, [ebp+var_8]
004055CC call    sub_401FAC
004055D1 mov     edx, [ebp+var_8]
004055D4 mov     eax, edi
004055D6 call    sub_402000
```

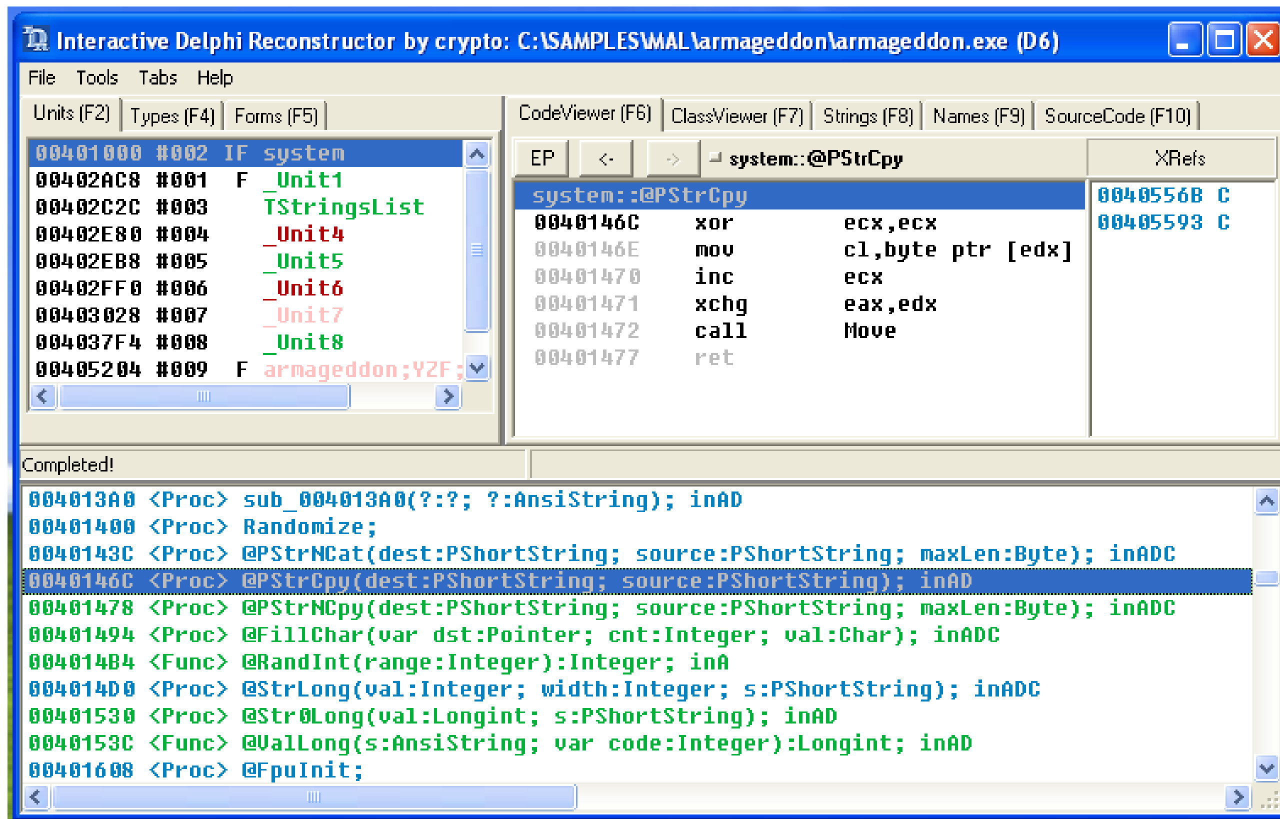
```
004055DB
004055DB loc_4055DB:
004055DB inc     ebx
004055DC dec     esi
004055DD jnz     loc_40555E
```

Graph overview

100.00% | (-22,782) | (542,545) | 00004914 | 00405514: sub_405514

Function `sub_405514()`

Sadly, this is par for the course with Delphi malware, where even the simplest of operations tend to be implemented through layer upon layer of calls through the framework's myriad of functions. That is where IDR comes in handy: to immediately identify as many of these built-in Delphi functions and class methods as possible. The process is straightforward: for each of the functions of interest, such as `sub_40146C()`, `sub_40143C()`, etc., we check IDR to see if it has automatically identified them as known Delphi functions. Sure enough, IDR comes to our aid and identifies `sub_40146C()` as the Delphi function `PStrCpy()`, which takes destination and source `ShortString` pointers in registers `EAX` and `EDX`, respectively:



IDR identification of `sub_40146C()` as Delphi function `PStrCpy()`

Back in IDA, we annotate `sub_40146C()` appropriately, and continue consulting IDR in order to identify any other Delphi routines in the primary loop block. Sometimes IDR cannot identify them all, in which case we have to manually figure out what they do, using the Delphi documentation as a guide. We make note of the fact that these string manipulation calls are operating on Delphi `ShortString` variables, which are simple little strings with a fixed buffer and a maximum size of 255 characters; their first byte stores their length, followed by their actual content buffer.

Working our way through `sub_405514()`, and annotating the code in IDA, yields the following:

IDA View-A

```
0040555E
0040555E loc_40555E:              ; skip every other iteration
0040555E test    bl, 1
00405561 jz      short loc_4055DB
```

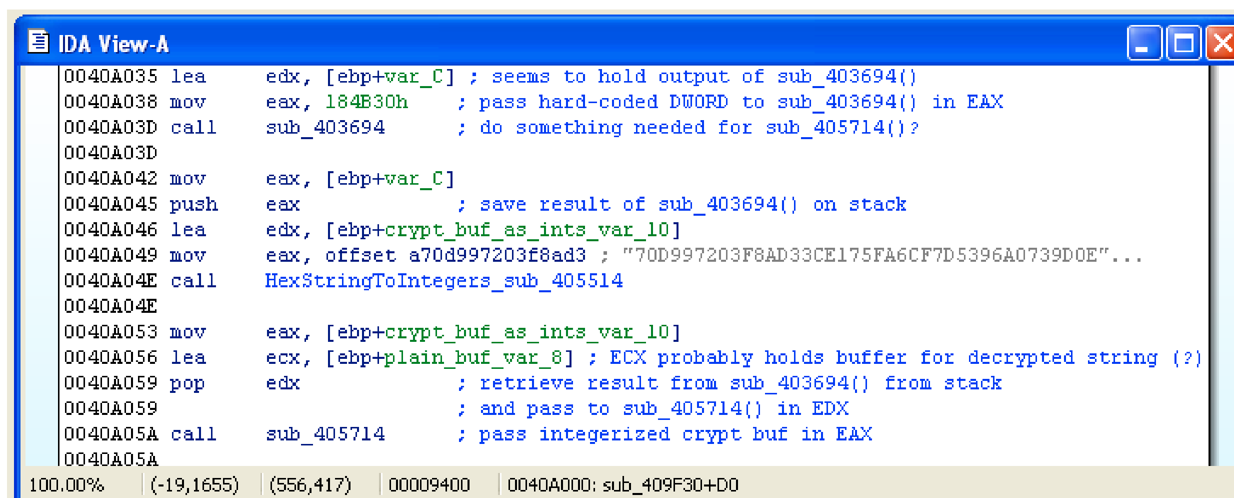
```
00405563 mov     edx, offset dword_40560C ; 0x01 0x24, or literal "$" ShortString
00405568 lea     eax, [ebp+shortstring_var_10]
0040556B call    PStrCpy_sub_40146C ; shortstring_var_10 := "$"
00405570 lea     eax, [ebp+shortstring_var_14]
00405573 mov     edx, [ebp+src_buf_var_4] ; src_buf_var_4 was copied from arg0 (passed in EAX)
00405576 mov     dl, [edx+ebx-1] ; EBX is our loop iterator, which we'll call 'k'
0040557A mov     [eax+1], dl     ; shortstring_var_14[1] = src_buf_var_4[k]
0040557D mov     byte ptr [eax], 1 ; set length of shortstring_var_14 to 1
00405580 lea     edx, [ebp+shortstring_var_14]
00405583 lea     eax, [ebp+shortstring_var_10]
00405586 mov     cl, 2
00405588 call    PStrNCat_sub_40143C ; shortstring_var_14 := "$" + src_buf[k]
0040558D lea     edx, [ebp+shortstring_var_10]
00405590 lea     eax, [ebp+shortstring_var_18]
00405593 call    PStrCpy_sub_40146C ; shortstring_var_18 := shortstring_var_10
00405598 lea     eax, [ebp+shortstring_var_14]
0040559B mov     edx, [ebp+src_buf_var_4]
0040559E mov     dl, [edx+ebx]
004055A1 mov     [eax+1], dl     ; shortstring_var_14[1] = src_buf_var_4[k+1]
004055A4 mov     byte ptr [eax], 1 ; set length of shortstring_var_14 to 1
004055A7 lea     edx, [ebp+shortstring_var_14]
004055AA lea     eax, [ebp+shortstring_var_18]
004055AD mov     cl, 3
004055AF call    PStrNCat_sub_40143C ; shortstring_var_18 := "$" + src_buf[k] + src_buf[k+1]
004055B4 lea     edx, [ebp+shortstring_var_18]
004055B7 lea     eax, [ebp+var_C]
004055BA call    CopyString_sub_401FEC ; string_var_C := "$" + src_buf[k] + src_buf[k+1]
004055BF mov     eax, [ebp+var_C]
004055C2 call    CallValLong_sub_403688 ; convert string to integer
004055C7 mov     edx, eax
004055C9 lea     eax, [ebp+var_8]
004055CC call    StringFromByte_sub_401FAC ; convert byte integer back to character
004055D1 mov     edx, [ebp+var_8]
004055D4 mov     eax, edi        ; EDI holds destination buffer (passed in EDX)
004055D6 call    StringConcat_sub_402000 ; add converted character to dest buf (EDI)
```

100.00% | (36,666) | (139,595) | 0000495E | 0040555E: sub_405514:loc_40555E

Function `sub_405514()` after annotation

The convoluted code basically iterates through the (encrypted) source string, originally passed in via the `EAX` register, character by character. However, it does nothing (skips the entire inner loop) on the odd iterations as per instruction `0x00405561`, so essentially it operates on two characters at a time. On every even iteration (when the 1-based counter `EBX` is a multiple of two), it painstakingly constructs a 3-character string that starts with the literal "`$`" and contains the next two characters in the source string; e.g., `$70` (for the first pair of characters.) It then calls the `ValLong()` procedure to convert each such little string into its integer equivalent, which corresponds to the byte value `112`, or `0x70`, in this case; the prefix "`$`" is Delphi notation for a string literal in hexadecimal format. It then appends each converted integer to the destination buffer, which was originally passed in via the `EDX` register.

So in other words, it turns out that `sub_405514()` is not really decrypting anything, but is just a rather roundabout (to say the least) procedure for converting our encrypted source string, expressed in hexadecimal format, into integer equivalents. So we'll go back to our initial starting point, where `sub_405514()` - which we now rename as `HexStringToIntegers_sub_405514()` - is being called with our original encrypted string in hex format:



Function sub_409F30() after annotation and speculative reversing

OK, since HexStringToIntegers_sub_405514() is just converting the encrypted string into integers, then perhaps the following call to sub_405714() is doing the actual decryption dirty work? Well, clearly the integerized result from HexStringToIntegers_sub_405514() is being passed via EAX, as well as another variable via ECX, which we'll optimistically assume is the destination buffer for the decrypted string. Furthermore, sub_405714() is expecting another argument via EDX; this was stored on the stack prior to calling since HexStringToIntegers() at instruction 0x0040A045. It looks like this value is the output, temporarily stored in var_C, of yet another function sub_403694(), which itself takes a 32-bit DWORD hard-coded as 0x00184B30.

This is starting to look promising: the majority of malware crypto mechanisms use some kind of key that is combined with the encrypted string via some custom algorithm that converts it into a plain text string. And now we are seeing that sub_405714() is being
passed the (integerized) encrypted string, a destination buffer, and some other (as yet unknown) piece of data output from sub_403694(); which could perhaps be a key string; this matches the general pattern that we would expect to find in a decryption routine. So we are going to tentatively speculate that sub_405714() is the actual decryption routine, and that sub_403694() derives the key, perhaps using the hard-coded value as 0x00184B30 as key-generating material.

So lets dive into sub_405714() with this working assumption in mind:

IDA View-A

```
00405714 sub_405714 proc near
00405714
00405714 key_buf_var_8= dword ptr -8
00405714 crypt_buf_var_4= dword ptr -4
00405714
00405714 push    ebp
00405715 mov     ebp, esp
00405717 add     esp, 0FFFFFFF8h
0040571A push    ebx
0040571B mov     ebx, ecx         ; store destination buffer in EBX
0040571D mov     [ebp+key_buf_var_8], edx ; store key buffer
00405720 mov     [ebp+crypt_buf_var_4], eax ; store encrypted source string
00405723 mov     eax, [ebp+crypt_buf_var_4]
00405726 call    sub_4021E0
0040572B mov     eax, [ebp+key_buf_var_8]
0040572E call    sub_4021E0
00405733 xor     eax, eax
00405735 push    ebp
00405736 push    offset sub_405779
0040573B push    dword ptr fs:[eax]
0040573E mov     fs:[eax], esp    ; set up exception handling
0040573E                          ;
00405741 lea     eax, [ebp+crypt_buf_var_4]
00405744 mov     edx, [ebp+key_buf_var_8]
00405747 call    sub_4056C8       ; *** do something with key and encrypted string
0040574C lea     eax, [ebp+crypt_buf_var_4]
0040574F call    sub_405610       ; *** post-processing of decrypted string??
00405754 mov     eax, ebx
00405756 mov     edx, [ebp+crypt_buf_var_4]
00405759 call    LStrAsg_sub_401EE4 ; copy decrypted version into destination buffer
0040575E xor     eax, eax
00405760 pop     edx
00405761 pop     ecx
00405762 pop     ecx
00405763 mov     fs:[eax], edx    ; clear exception handling
00405766 push    offset loc_405780
```

```
0040576B
0040576B loc_40576B:
0040576B lea     eax, [ebp+key_buf_var_8]
0040576E mov     edx, 2
00405773 call    FreeString_sub_401EB4
00405778 retn
00405778 sub_405714 endp ; sp = -14h
00405778
```

100.00% | (-38,80) | (20,665) | 00004B4F | 0040574F: sub_405714+3B

Function sub_405714()

Function sub_405714() is pretty simple: it stores the source encrypted string and (presumed) key string into stack local variables crypt_buf_var_4 and key_buf_var_8, respectively, and then sets up some exception handling; the function then boils down to calling two routines: passing the encrypted and key buffers to sub_4056C8(), and then passing this (freshly decrypted?) buffer to sub_405610(), perhaps for some post-processing, after which the decrypted string is presumably copied into the destination buffer. This structure suggests that the encrypted buffer is decryptein-place

IDA View-A

```
CODE:004056C8 sub_4056C8      proc near               ; CODE XREF: sub_405714+33↓p
CODE:004056C8                 push    esi
CODE:004056C9                 push    edi
CODE:004056CA                 push    ebx
CODE:004056CB                 or      eax, eax
CODE:004056CD                 jz      short DONE_loc_405710 ; if crypt buffer is NULL then nothing to do
CODE:004056CF                 push    eax
CODE:004056D0                 push    edx
CODE:004056D1                 call    UniqueString_sub_40223C ; forces a reference count of 1
CODE:004056D6                 pop     edx
CODE:004056D7                 pop     eax
CODE:004056D8                 mov     edi, [eax]      ; store crypt_buf[0] in EDI
CODE:004056DA                 or      edi, edi
CODE:004056DC                 jz      short DONE_loc_405710 ; if crypt_buf[0] is NULL (empty string) then done
CODE:004056DE                 mov     ecx, [edi-4]    ; iterate (via ECX) over length of crypt_buf
CODE:004056E1                 jecxz   short DONE_loc_405710 ; if len(crypt_buf) is 0 then done
CODE:004056E3                 mov     esi, edx        ; store key_buf in ESI
CODE:004056E5                 or      esi, esi
CODE:004056E7                 jz      short DONE_loc_405710 ; if key_buf is NULL then done
CODE:004056E9                 mov     edx, [esi-4]
CODE:004056EC                 dec     edx
CODE:004056ED                 js      short DONE_loc_405710 ; if len(key_buf) is 0 then done
CODE:004056EF                 mov     ebx, edx        ; EBX := len(key_buf) - 1
CODE:004056F1                 mov     ah, dl          ; initialize accumulator AH
CODE:004056F3                 cld                     ; make string operations go forward
CODE:004056F4
CODE:004056F4 LOOP_START_loc_4056F4:                  ; CODE XREF: sub_4056C8+46↓j
CODE:004056F4                 test    ah, 8           ; modify accummulator if its bit 3 is not set
CODE:004056F7                 jnz     short loc_4056FC
CODE:004056F9                 xor     ah, 1
CODE:004056FC
CODE:004056FC loc_4056FC:                             ; CODE XREF: sub_4056C8+2F↑j
CODE:004056FC                 not     ah              ; flip all bits of accummulator
CODE:004056FE                 ror     ah, 1           ; right shift accummulator by one
CODE:00405700                 mov     al, [ebx+esi]   ; pull next key byte
CODE:00405703                 xor     al, ah          ; XOR key byte against accummulator
CODE:00405705                 xor     al, [edi]       ; XOR next source byte against modified key byte
CODE:00405707                 stosb                   ; overwrite current source byte and advance EDI
CODE:00405708                 dec     ebx             ; consume key string backwards
CODE:00405709                 jns     short loc_40570D
CODE:0040570B                 mov     ebx, edx        ; cycle back to end of key string
CODE:0040570D
CODE:0040570D loc_40570D:                             ; CODE XREF: sub_4056C8+41↑j
CODE:0040570D                 dec     ecx             ; keep looping til source string is processed
CODE:0040570E                 jnz     short LOOP_START_loc_4056F4
CODE:00405710
CODE:00405710 DONE_loc_405710:                        ; CODE XREF: sub_4056C8+5↑j
CODE:00405710                                         ; sub_4056C8+14↑j ...
CODE:00405710                 pop     ebx
CODE:00405711                 pop     edi
CODE:00405712                 pop     esi
CODE:00405713                 retn
CODE:00405713 sub_4056C8      endp
```

00004B05 | 00405705: sub_4056C8+3D

Function sub_4056C8()

After doing some error checking for empty source and/or key strings, the decryption routine operates in a single loop over the length of the encrypted source string. The core of the decryption operations take place in the loop block between instructions `0x004056F4` and `0x0040570E`. The malware uses up the key string backwards from end to beginning, cycling back to the end as many times as needed to process the source string, using `EBX` as an indexing register.

The code also uses a 1-byte "accummulator" variable, implemented with register `AH`, that gets initialized at `0x004056F1` as one less than the length of the key string. On each iteration of the loop, it gets updated by flipping all its bits, bit rotating it one bit to the right, and flipping its least significant bit if and only if its fourth least significant bit is set; it is then `XOR`ed against the current key byte, and finally the result is then `XOR`ed against the current source byte. This routine implements a stream cipher, as opposed to a straight substitution table, in the sense that a particular encrypted byte can get mapped to different plain text bytes depending upon the position within the encrypted string at which it appears - even when the key string is held constant.

So now that we have reversed the algorithm used by `sub_4056C8()` to apply the key string to the encrypted string, it is trivial to write a Python decryptor script to duplicate the process. However, we still need to go back and figure out exactly what kind of post-processing the second function, `sub_405610()`, does to the decrypted string:

```
CODE:00405620                 or      esi, esi        ; ESI now points to mesg_buf
CODE:00405622                 jz      short DONE_loc_405661 ; if mesg_buf is empty then done
CODE:00405624                 mov     ecx, [esi-4]    ; extract len(mesg_buf) into ECX
CODE:00405627                 jecxz   short DONE_loc_405661 ; if zero-length mesg buf then done
CODE:00405629                 mov     edx, ecx        ; initialize accummulator (EDX)
CODE:0040562B                 lea     edi, byte_405665 ; EDI points to substitution table
CODE:00405631                 cld                     ; make string ops go forward
CODE:00405632
CODE:00405632 LOOP_START_loc_405632:                  ; CODE XREF: sub_405610+4F↓j
CODE:00405632                 xor     eax, eax
CODE:00405634                 lodsb                   ; move next byte from mesg_buf (ESI) into AL
CODE:00405635                 sub     ax, 20h         ; subtract 0x20 from mesg byte
CODE:00405639                 js      short END_DESCRAMBLE_loc_40565E
CODE:0040563B                 cmp     ax, 5Fh         ; only descramble if 0x20 <= mesg byte <= 0x7F
CODE:0040563F                 jg      short END_DESCRAMBLE_loc_40565E
CODE:00405641                 mov     ebx, eax        ; keep value of EAX to be used @ 0x405654
CODE:00405643                 test    cx, 3           ; modify accummulator EDX depending on iter index
CODE:00405648                 jz      short END_BITSHIFT_loc_40564D
CODE:0040564A                 rol     edx, 3
CODE:0040564D
CODE:0040564D END_BITSHIFT_loc_40564D:                ; CODE XREF: sub_405610+38↑j
CODE:0040564D                 and     dl, 1Fh         ; update accummulator EDX some more
CODE:00405650                 xor     al, dl          ; apply accummulator to modified mesg byte
CODE:00405652                 add     edx, ecx        ; update accummulator (EDX)
CODE:00405654                 add     edx, ebx        ; EBX saved @ 0x405641
CODE:00405656                 mov     al, [eax+edi]   ; choose new byte value from subst table
CODE:00405659                 add     al, 20h         ;   based on original mesg byte value
CODE:0040565B                 mov     [esi-1], al     ; overwrite new value for current mesg byte
CODE:0040565E
CODE:0040565E END_DESCRAMBLE_loc_40565E:              ; CODE XREF: sub_405610+29↑j
CODE:0040565E                                         ; sub_405610+2F↑j
CODE:0040565E                 dec     ecx             ; iterate over each byte in mesg_buf
CODE:0040565F                 jnz     short LOOP_START_loc_405632
CODE:00405661
CODE:00405661 DONE_loc_405661:                        ; CODE XREF: sub_405610+5↑j
```

Function `sub_405610()`

The function takes only a single argument: the message buffer containing the result of the decryption process implemented in `sub_4056C8()`; there appears to be no key string involved this time. After performing the obligatory checks for a `NULL` or empty message buffer, the function iterates over each byte in this message buffer, which is addressed by `ESI`. The loop uses another "accummulator" variable, stored in `EDX`, which is initialized to be the length of the message buffer string. It also appears to use yet another array of 96 bytes hard-coded at location `0x00405665`; investigating this array, we see that it looks like this:

```
CODE:00405665 byte_405665     db 90, 84, 91, 92, 85, 78, 72, 79 ; DATA XREF: sub_405610+1B↑o
CODE:00405665                 db 86, 93, 94, 87, 80, 73, 66, 60
CODE:00405665                 db 67, 74, 81, 88, 95, 89, 82, 75
CODE:00405665                 db 68, 61, 54, 48, 55, 62, 69, 76
CODE:00405665                 db 83, 77, 70, 63, 56, 49, 42, 36
CODE:00405665                 db 43, 50, 57, 64, 71, 65, 58, 51
CODE:00405665                 db 44, 37, 30, 24, 31, 38, 45, 52
CODE:00405665                 db 59, 53, 46, 39, 32, 25, 18, 12
CODE:00405665                 db 19, 26, 33, 40, 47, 41, 34, 27
CODE:00405665                 db 20, 13, 6, 0, 7, 14, 21, 28
CODE:00405665                 db 35, 29, 22, 15, 8, 1, 2, 9
CODE:00405665                 db 16, 23, 17, 10, 3, 4, 11, 5
CODE:004056C5                 db 0C3h ; +
CODE:004056C6                 db  8Bh ; ï
CODE:004056C7                 db 0C0h ; +
```

Array of 96 bytes at 0x00405665

Hmmm, that is interesting: this 96-byte array seems to contain each of the 96 byte values between 0 and 96 in some kind of scrambled order; this looks like it could be a substitution table.

Returning to the loop block in function sub_405610(), we see that it starts by using the lodsb instruction to pull the next byte from the message buffer (in ESI) and advance that buffer by one byte. It then skips the remainder of the loop if the value of this message byte lies outside the range of 0x20 (space character) to 0x7F (lower-case z); that range happens to contain 96 ASCII characters - the same number of characters in the substitution table. Coincidence? Probably not.

If the current message byte does happen to lie within this 96-character range, then the malware will do some shenanigans to update the value of the accumulator in EDX; specifically, if the fourth least significant bit in the backward loop counter ECX is set, then it will apply a bitwise rotation of three bits to the accumulator, followed by clearing all but the five least significant bits. It then XORs this accumulator value against the current message byte value (after subtracting 0x20 from that value), followed by yet more updates to the accumulator: increasing it by the sum of the backward loop counter and the pre-XORed message byte value; this ensures that the accumulator vigorously changes its value with each iteration of the loop. Finally, the post-XORed message byte value is used as an index into the 96-element substitution table at location 0x00405665 in order to pull out a final ASCII character in the range of 0x20 to 0x7F, which is used to overwrite the original element of the message buffer string. In other words, the "post-processing" done by function sub_405610() is

actually an entire second round of decryption. Again, it will be trivial to implement this second decryption round in Python.

To make a long story short, we now see that this new version of Armageddon uses a two-stage crypto mechanism in which the first stage, implemented by `sub_4056C8()`, uses a custom stream cipher that applies a key string to partially decrypt the original encrypted string in place, and the second stage, implemented by function `sub_405610()`, performs a stream-based descrambling operation that makes use of a hard-coded 96-byte substitution table. Both stages have an extra "moving part" (the accummulators) that were perhaps implemented in order to resist automated cryptanalysis.

So now that we have fully reversed the crypto algorithm, we just need to know what key string is used by this particular malware sample in order to decrypt the strings. This is where we need to return all the way back to the vicinity of `0x0040A03D`, shortly before the `HexStringToInteger()` call; this is where the key string appears to be generated in the call to `sub_403694()`. Recall that this function is passed two arguments: a 32-bit `DWORD` integer value `0x00184B30`, and a destination buffer into which the decryption key string will presumably be written. Once again, IDR assists us by identifying the sub-functions called by `sub_403694()`:

```
00403694
00403694
00403694
00403694 GenerateKeyString_sub_403694 proc near
00403694
00403694 string_repr_var_100= dword ptr -100h
00403694
00403694 push    ebx
00403695 push    esi
00403696 add     esp, 0FFFFFEF4h ; alloc 268 bytes on stack
0040369C mov     esi, edx        ; ESI := destination buffer
0040369E mov     ebx, eax        ; EAX used to pass 32-bit integer
004036A0 lea     edx, [esp+10Ch+string_repr_var_100]
004036A4 mov     eax, ebx
004036A6 call    Str0Long_sub_401530 ; convert 32-bit integer into string representation
004036A6
004036AB lea     edx, [esp+10Ch+string_repr_var_100]
004036AF mov     eax, esp
004036B1 mov     cl, 0Bh
004036B3 call    PStrNCpy_sub_401478 ; copy up to 11 bytes into stack buffer
004036B3
004036B8 mov     eax, esi
004036BA mov     edx, esp
004036BC call    CopyString_sub_401FEC ; copy stack variable into dest buffer
004036BC
004036C1 add     esp, 10Ch
004036C7 pop     esi
004036C8 pop     ebx
004036C9 retn
004036C9
004036C9 GenerateKeyString_sub_403694 endp
004036C9
```

Function `GenerateKeyString_sub_403694()`

The critical call is to the Delphi procedure `PStr0Long()`, which takes a 32-bit integer and converts it into its corresponding ASCII string representation. In our case, the first argument `0x00184B30` (or `1592112` in decimal) would get converted into the key string `"1592112"`. The remainder of function `sub_403694()` just clumsily copies the resulting string into the destination buffer in Delphi's excruciatingly roundabout way. It is now clear that a 32-bit `DWORD` (e.g., `0x00184B30`) is used as the underlying key material, which is simply expanded by function `sub_403694()` into a proper ASCII key string for use by the actual decryption routine.

We now have everything we need to write a decryption tool to expose Armageddon's secrets. A simple Python implementation is as follows:

```
# Armageddon Decryptor
# Copyright (c) 2012 Arbor Networks

import sys

def ror1(value): # ROR 1
    return ((value & 0x01) << 7) | ((value & 0xfe) >> 1)
def rol3(value): # ROL 3
    return ((value & 0x1fffffff) << 3) | ((value & 0xe0000000) >> 29)

RAW_SCRAMBLE_TABLE = \
        "5A545B5C554E484F565D5E575049423C" \
        "434A51585F59524B443D3630373E454C" \
        "534D463F38312A242B32394047413A33" \
        "2C251E181F262D343B352E272019120C" \
        "131A21282F29221B140D0600070E151C" \
        "231D160F0801020910171110A03040B05"

def decrypt_armageddon2(ciphertext, keystring):
    cipherbytes = [int(ciphertext[k*2:k*2+2], 16) \
                   for k in range(len(ciphertext) // 2)]

    # Stage I. Apply decryption key
    start_idx = len(keystring) - 1
    key_idx = start_idx
    accum = start_idx
    for k, cipherbyte in enumerate(cipherbytes):
        if not (accum & 0x8):
            accum ^= 0x1
        accum = ror1(~accum % 256)
        cipherbytes[k] = (ord(keystring[key_idx]) ^ accum ^ cipherbyte)
        key_idx = start_idx if key_idx == 0 else key_idx - 1
```

```
    # Stage II. Apply scrambling table
    scramble_table = [int(RAW_SCRAMBLE_TABLE[k*2:k*2+2], 16) \
            for k in range(len(RAW_SCRAMBLE_TABLE) // 2)]
    idx = len(cipherbytes)
    accum = idx
    for k, cipherbyte in enumerate(cipherbytes):
        orig_val = cipherbyte - 0x20
        if orig_val <= 0x5f:
            ebx = orig_val
            if (idx & 0x3):
                accum = rol3(accum)
        accum &= 0xffffff1f
        orig_val = ((orig_val & 0xffffff00) | ((orig_val & 0xff) ^ (accum &
0xff)))
        accum += (idx + ebx)
        cipherbytes[k] = (scramble_table[orig_val] + 0x20)
        idx -= 1
    return cipherbytes

# Example: python decrypt.py 890D9F1363245A5D 1B669
if __name__ == '__main__':
    if len(sys.argv) != 3:
        print "usage: %s CIPHERTEXT KEYCODE" % sys.argv[0]
        sys.exit(1)
    ciphertext = sys.argv[1]
    # Convert integer key material into key string
    keystring = "%d" % int(sys.argv[2], 16)
    plainbytes = decrypt_armageddon2(ciphertext, keystring)
    plaintext = ''.join([chr(plainbyte) for plainbyte in plainbytes])
    print plaintext
```

Python script to decrypt Armageddon strings

Before running this script, we'll do a quick scan of the binary to enumerate which 32-bit value(s) our Armageddon sample uses for key material; e.g., which values are passed via `EAX` in the various invocations of `sub_403694()`. We find that the vast majority of its encrypted strings use the value `0x001B669` (i.e., `112233` in decimal) as key material, although a handful of the strings turned out to be encrypted using the aforementioned value `0x00184B30`.
We'll now run our Python script against each of the 45 strings in our sample (actually, we automate this process as well so that we can apply it against all of our other Armageddon samples too.) This yields the following loot:

```
headers=UserAgent
headers=Referer
headers=Accept
headers=AcceptLanguage
headers=AcceptEncoding
://
href="
http://
src="
.scanflood=
.timeoutflood
.downloadflood
User-Agent:
Referer:
GET
.apacheflood
Range: bytes=0-,5-0,5-1,5-2,5-3,5-4,<...>,5-1299,5-1300
Accept-Encoding: gzip
HEAD
.vbulletinflood
Accept:
Accept-Language:
Accept-Encoding:
```

```
application/x-www-form-urlencoded
POST
.phpbbflood
.ontcpflood
.onudpflood
.timeoutflood=
.scanflood=
id=
:*:
Shell
Software
SYSTEM\ControlSet001\Services\SharedAccess\Parameters\FirewallPolicy\Standard
     Profile\AuthorizedApplications\List\
SYSTEM\ControlSet002\Services\SharedAccess\Parameters\FirewallPolicy\Standard
     Profile\AuthorizedApplications\List\
SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\Stan
     dardProfile\AuthorizedApplications\List\
Software\Microsoft\Windows\CurrentVersion\Run\
Software\Microsoft\Windows NT\CurrentVersion\Winlogon
Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
APPDATA
.exe
http://testse.in/sdfghj/cmd.php
http://kgjghh.in/sdfghj/cmd.php
http://lyam2zw.in/sdfghj/cmd.php
```

Decrypted versions of sensitive Armageddon strings

Upon first glance, it would appear that Armageddon's encrypted strings include the following goodies:

- The names of the DDoS attack commands to which the bot responds (e.g., `.timeoutflood`, `.onudpflood`, etc.);
- The Windows Registry keys it uses for installing itself;

- The URLs of its three C&C servers (these happen to be the only ones that were encrypted using `0x00184B30` as the key material);
- The various components used by the bot to generate its DDoS attack traffic (e.g., `User-Agent:`, `Referer:`, etc.);
- The request strings it sends back to its C&C in order to download string sets for use in randomizing HTTP headers during DDoS floods (e.g., `headers=UserAgent`, `headers=Referer`, etc.)

Further analysis of the bot binary in IDA Pro confirmed these findings, and allowed us to study and understand the exact nature of the traffic generated by Armageddon's various DDoS attack routines; this of course is our primary motivation for breaking malware crypto systems: so that we can develop network defenses that protect potential victim sites from attack by unsavory characters such as Armageddon.

In addition to a generic HTTP flood attack and some old-school volumetric attacks such as TCP and UDP floods, it turns out that Armageddon has developed a variety of specialized HTTP flooding techniques that appear to be customized to target different types of victim websites; the names of the associated attack commands provide clues as to the nature of their intended targets: `.vbulletinflood`, `.phpbbflood`, and `.apacheflood`.
This post is already quite lengthy so we will not discuss at this time the technical properties of the network traffic generated by these various Armageddon attack modes. However, we will mention that the implementation of the `.apacheflood` command was of particular interest; it makes use of the following (decrypted) string when formulating its flooding requests:

```
Range: bytes=0-,5-0,5-1,5-2,5-3,5-4,<...>,5-1299,5-1300
```

This string represents an optional HTTP header that turns out to be included in DDoS flooding requests generated by the bot when performing an `.apacheflood` attack; this string, along with another encrypted Armageddon string, `Accept-Encoding: gzip`, have been associated with the so-called "Kill Apache" attack, a type of highly assymetric low-bandwidth DDoS technique that has emerged relatively recently.
In a nutshell, the Kill Apache attack abuses the HTTP protocol by requesting that the target web server return the requested URL content in a huge number of individual chunks, or byte ranges. This can cause a surprisingly heavy load on the target server; in particular, certain versions of the Apache HTTP server handle such requests extremely poorly and in some cases can be brought to their knees by a single attacking client. To our knowledge, this is the first time that the Kill Apache attack has reared its ugly head in actual botnet code in the wild, as opposed to proof-of-concept and/or standalone attack tools.

Of course, once we have taken the liberty of prying open Armageddon's kimono, it was straightforward to write a "fake Armageddon" client that phones home to the (now decrypted) C&C URLs and engages in communication that impersonates a real bot. This allows us to gather additional intelligence on the activities and behavioral patterns of Armageddon; in particular, we can now monitor the various Armageddon botnets to log the targets that it attacks, and the types of DDoS floods uses in those attacks. Among other things, this technique allowed us to discover that at least one of the botnets powered by the most recent Armageddon code base took part in the DDos attacks related to the recent Russian election in early December. We will continue to keep a watchful eye on Armageddon going forward.