



2009

A Tales of Reversing & Keygenning Two MD5 Registration Schemas



Author: 2kAD, Nieylana

Graphic Editor: Shub-Nigurrath

ARTeam

February 2009

DISCLAIMER

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this tutorial have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched by other fellows, and cracked versions were available since a lot of time. ARTeam or the authors of the papers shouldn't be considered responsible for damages to the companies holding rights on those programs. The scope of this document as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application.

VERIFICATION

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: <http://releases.accessroot.com>



TABLE OF CONTENTS

Disclaimer.....	2
Verification	2
1 KEYGENNING MD5 – 2KAD	4
1.1 Forewords	4
1.1.1 Tools used:.....	4
1.2 Before we start.....	4
1.3 Starting up and getting ready	7
1.4 Analyzing the algorithm	10
1.5 Figuring out a valid key.....	19
1.6 Reversing the algorithm	24
1.7 Final words.....	24
2 KEYGENNING MD5, NIEYLANA	25
2.1 Abstract.....	25
2.2 Target.....	25
2.3 Analyzing Target.....	25
2.4 Finding the Verification Routine.....	26
2.5 Finding the Beginning of the MD5 Routine.....	27
2.6 Finding the String Being Hashed	28
2.7 Use of the MD5 Hash.....	28
2.8 Summary.....	29
2.9 Greetings.....	30
2.10 Conclusion.....	30

1 KEYGENNING MD5 – 2KAD

1.1 FOREWORDS

I have been on the computer scene for a long time... Since 1984 to be exact, and have been reversing or cracking (so it was called back then) since that time. I have seen many protections come and go, and I have unpacked pretty much every protection known to man. However, keygenning is new to me.... Or serious keygenning is. In this tutorial I will show you how I reversed what I will describe as my first serious keygen. The tutorial is not a complete walk through, because it would take months to write. It's written so that reversers with some knowledge should be able to see how I figured out this protection. If you are a complete newbie it's not a tutorial for you yet.

1.1.1 TOOLS USED:

A PC, Olly debugger v1.10, a working brain.

1.2 BEFORE WE START...

In the following tutorial you will see my comments in the disassembly, that the target uses MD5 hashing. But how do I know ? Well, of course one will have to trace the target and eventually you will stumble upon certain code snippets that for sure will give you the answer.

Here is the first clue:

```

00CFAS40 8B4424 04 MOV EAX,DWORD PTR SS:[ESP+4]
00CFAS44 33C9 XOR ECX,ECX
00CFAS46 C700 0123456 MOV DWORD PTR DS:[EAX],67452301
00CFAS4C C740 04 89AB MOV DWORD PTR DS:[EAX+4],EFCDA889
00CFAS53 C740 08 FEDC MOV DWORD PTR DS:[EAX+8],98BADCFE
00CFAS5A C740 0C 7654 MOV DWORD PTR DS:[EAX+C],10325476
00CFAS61 8948 10 MOV DWORD PTR DS:[EAX+10],ECX
00CFAS64 8948 14 MOV DWORD PTR DS:[EAX+14],ECX
00CFAS67 8948 58 MOV DWORD PTR DS:[EAX+58],ECX
00CFAS6A C3 RETN
00CFAS6B 00

```

This taken from RFC 1351 (<http://tools.ietf.org/html/rfc1321>):

```

/* MD5 initialization. Begins an MD5 operation, writing a new context.
 */
void MD5Init (context)
MD5_CTX *context;                                /* context */
{
    context->count[0] = context->count[1] = 0;
    /* Load magic initialization constants.
 */
    context->state[0] = 0x67452301;
    context->state[1] = 0xefcdab89;
    context->state[2] = 0x98badcfe;
    context->state[3] = 0x10325476;
}

```

Here is the final proof:

00CFA570	885424 0C	MOV EDX,DWORD PTR SS:[ESP+C]	
00CFA574	8B4C24 04	MOV ECX,DWORD PTR SS:[ESP+4]	
00CFA578	53	PUSH EBX	
00CFA579	8BC2	MOV EAX,EDX	
00CFA57B	8B59 08	MOV EBX,DWORD PTR DS:[ECX+8]	
00CFA57E	55	PUSH EBP	
00CFA57F	8B69 04	MOV EBP,DWORD PTR DS:[ECX+4]	
00CFA582	4A	DEC EDX	
00CFA583	56	PUSH ESI	
00CFA584	8B71 0C	MOV ESI,DWORD PTR DS:[ECX+C]	
00CFA587	85C0	TEST EAX,EAX	
00CFA589	0F84 C606000	JE 00CFAC55	
00CFA58F	8B4424 14	MOV EAX,DWORD PTR SS:[ESP+14]	
00CFA593	57	PUSH EDI	
00CFA594	83C0 38	ADD EAX,38	
00CFA597	42	INC EDX	
00CFA598	895424 18	MOV DWORD PTR SS:[ESP+18],EDX	
00CFA59C	8B78 C8	MOV EDI,DWORD PTR DS:[EAX-38]	
00CFA59F	8BD6	MOV EDX,ESI	
00CFA5A1	33D3	XOR EDX,EBX	
00CFA5A3	23D5	AND EDX,EBP	
00CFA5A5	33D6	XOR EDX,ESI	
00CFA5A7	03D7	ADD EDX,EDI	
00CFA5A9	8BFB	MOV EDI,EBX	
00CFA5AB	8BF2	MOV ESI,EDX	
00CFA5AD	8B11	MOV EDX,DWORD PTR DS:[ECX]	
00CFA5AF	33FD	XOR EDI,EBP	
00CFA5B1	8D9416 78A46	LEA EDX,DWORD PTR DS:[ESI+EDX+D76AA478]	Look at this
00CFA5B8	8B70 CC	MOV ESI,DWORD PTR DS:[EAX-34]	
00CFA5BB	C1C2 07	ROL EDX,7	
00CFA5BE	03D5	ADD EDX,EBP	
00CFA5C0	23FA	AND EDI,EDX	
00CFA5C2	33FB	XOR EDI,EBX	
00CFA5C4	03FE	ADD EDI,ESI	
00CFA5C6	8B71 0C	MOV ESI,DWORD PTR DS:[ECX+C]	
00CFA5C9	8D8437 56B7C	LEA ESI,DWORD PTR DS:[EDI+ESI+E8C7B756]	Look at this
00CFA5D0	8BFD	MOV EDI,EBP	
00CFA5D2	C1C6 0C	ROL ESI,0C	
00CFA5D5	03F2	ADD ESI,EDX	
00CFA5D7	33FA	XOR EDI,EDX	
00CFA5D9	23FE	AND EDI,ESI	
00CFA5DB	33FD	XOR EDI,EBP	
00CFA5DD	8B68 D0	MOV EBP,DWORD PTR DS:[EAX-30]	
00CFA5E0	03FD	ADD EDI,EBP	
00CFA5E2	8BEE	MOV EBP,ESI	
00CFA5E4	33EA	XOR EBP,EDX	
00CFA5E6	8D8C1F DB70B	LEA EDI,DWORD PTR DS:[EDI+EBX+242070DB]	Look at this
00CFA5E8	03F2 04	ADD EDI,EBP	

Again this taken from RFC 1351:

```
/* MD5 basic transformation. Transforms state based on block.
 */
static void MD5Transform (state, block)
UINT4 state[4];
unsigned char block[64];
{
    UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];

    Decode (x, block, 64);

    /* Round 1 */
    FF (a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
    FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
    FF (c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
    FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
    FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
    FF (d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
    FF (c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */
    FF (b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */
    FF (a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */
    FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */
    FF (c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
    FF (b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
    FF (a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
    FF (d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
    FF (c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
    FF (b, c, d, a, x[15], S14, 0x49b40821); /* 16 */
}
```

It's evident that we're in the first round of MD5. Also you can use IDA findcrypt plugin. This will identify it for you. There are other tools out there that will do the same job, but being able to identify MD5 yourself will save you time.

1.3 STARTING UP AND GETTING READY

Execute “drag2pdf.exe” and go to the registration dialog. Below you will see what I typed in. Don’t mind the strange language displayed in the picture, it’s just my native language.



Once you typed in a name and serial, press OK.

You will get a messagebox saying the that serial is not valid. So I decided to set breakpoints on **MessageBoxA**, **MessageBoxExA**, **MessageboxExW** and run it again.

We trap it at **MessageboxExW**. Back trace until you end up in a code-section.

00C72B76	. 8D4D EC	LEA ECX,DWORD PTR SS:[EBP-14]	
00C72B79	. 8845 F3	MOV BYTE PTR SS:[EBP-D],AL	
00C72B7C	. 6A 11	PUSH 11	
00C72B7E	. E8 BEBB1600	CALL 00DDE741	[Arg1 = 00000011 edocpdfp.00E1E741
00C72B83	. 399E AC000000	CMP DWORD PTR DS:[ESI+AC],EBX	
00C72B89	~ 76 6D	JBE SHORT 00C72BF8	From here
00C72B8B	. 385D F3	CMP BYTE PTR SS:[EBP-D],BL	
00C72B8E	~ 74 68	JE SHORT 00C72BF8	From here
00C72B90	. 8B86 C4000000	MOV EAX,DWORD PTR DS:[ESI+C4]	
00C72B96	. 3D 88000000	CMP EAX,88	
00C72B9B	~ 74 0E	JE SHORT 00C72BAB	
00C72B9D	. 3D 80160000	CMP EAX,1680	
00C72BA2	~ 75 54	JNZ SHORT 00C72BF8	From here
00C72BA4	. 68 FD000000	PUSH 0FD	
00C72BA9	~ EB 45	JMP SHORT 00C72BF0	
00C72BAB	> 68 FE000000	PUSH 0FE	
00C72BB0	~ EB 3E	JMP SHORT 00C72BF0	
00C72BB2	> 8B45 B8	MOV EAX,DWORD PTR SS:[EBP-48]	
00C72BB5	. 3BC3	CMP EAX,EBX	
00C72BB7	~ 75 05	JNZ SHORT 00C72BBE	
00C72BB9	. B8 EC88E000	MOV EAX,00E088EC	
00C72BBE	> 50	PUSH EAX	
00C72BBF	. 6A 06	PUSH 6	[Arg1 = 00000006 edocpdfp.00CA1347
00C72BC1	. E8 81E7FEFF	CALL 00C61347	
00C72BC6	. 59	POP ECX	
00C72BC7	. 8BF8	MOV EDI,EAX	
00C72BC9	. 59	POP ECX	
00C72BCA	. E8 29990500	CALL 00CCC4F8	
00C72BCF	. 85C0	TEST EAX,EAX	
00C72BD1	~ 75 39	JNZ SHORT 00C72C0C	
00C72BD3	. 81FF 37300000	CMP EDI,3037	
00C72BD9	~ 74 10	JE SHORT 00C72BEB	
00C72BDB	. 81FF 38300000	CMP EDI,3038	
00C72BE1	~ 74 08	JE SHORT 00C72BEB	
00C72BE3	. 81FF 3E300000	CMP EDI,303E	
00C72BE9	~ 75 21	JNZ SHORT 00C72C0C	
00C72BEB	> 68 CE000000	PUSH 0CE	
00C72BF0	> 8D4D EC	LEA ECX,DWORD PTR SS:[EBP-14]	
00C72BF3	. E8 49BB1600	CALL 00DDE741	[edocpdfp.00E1E741
00C72BF8	> 6A 30	PUSH 30	
00C72BFA	. 8BCE	MOV ECX,ESI	
00C72BFC	. FF75 E8	PUSH DWORD PTR SS:[EBP-18]	
00C72BFF	. FF75 EC	PUSH DWORD PTR SS:[EBP-14]	
00C72C02	. E8 339E1600	CALL 00DDCA9A	
00C72C07	~ E9 95000000	JMP 00C72CA1	
00C72C0C	> 6A 10	PUSH 10	[Arg1 = 00000010
00C72C0E	. 8D4D EC	LEA ECX,DWORD PTR SS:[EBP-14]	
00C72C11	. E8 29990500	CALL 00CCC4F8	[edocpdfp.00E1E741

Jumps from 00C72B89, 00C72B8E, 00C72BA2

Notice that the BAD BOY call originates from either 3 positions. So lets back trace a little until we hit gold.

00C72B38	. 8D45 D4	LEA EAX,DWORD PTR SS:[EBP-2C]	
00C72B3B	. 50	PUSH EAX	
00C72B3C	. 8D45 C4	LEA EAX,DWORD PTR SS:[EBP-3C]	
00C72B3F	. 50	PUSH EAX	
00C72B40	. E8 EEEBFEFF	CALL 00C61733	Let the fun begin
00C72B45	. 83C4 20	ADD ESP,20	
00C72B48	EB 14	JMP SHORT 00C72B5E	
00C72B4A	> 8D45 B4	LEA EAX,DWORD PTR SS:[EBP-4C]	
00C72B4D	. 50	PUSH EAX	
00C72B4E	. 8D45 D4	LEA EAX,DWORD PTR SS:[EBP-2C]	
00C72B51	. 50	PUSH EAX	
00C72B52	. 8D45 C4	LEA EAX,DWORD PTR SS:[EBP-3C]	
00C72B55	. 50	PUSH EAX	
00C72B56	. E8 7EE8FEFF	CALL 00C613D9	
00C72B5B	. 83C4 0C	ADD ESP,0C	
00C72B5E	> 3AC3	CMP AL,BL	
00C72B60	75 50	JNZ SHORT 00C72BB2	
00C72B62	. 8D45 B4	LEA EAX,DWORD PTR SS:[EBP-4C]	
00C72B65	. 50	PUSH EAX	
00C72B66	. 8D45 D4	LEA EAX,DWORD PTR SS:[EBP-2C]	
00C72B69	. 50	PUSH EAX	
00C72B6A	. 8D45 C4	LEA EAX,DWORD PTR SS:[EBP-3C]	
00C72B6D	. 50	PUSH EAX	
00C72B6E	. E8 66E8FEFF	CALL 00C613D9	
00C72B73	. 83C4 0C	ADD ESP,0C	
00C72B76	. 8D4D EC	LEA ECX,DWORD PTR SS:[EBP-14]	
00C72B79	. 8845 F3	MOV BYTE PTR SS:[EBP-D],AL	
00C72B7C	. 6A 11	PUSH 11	
00C72B7E	. E8 BEBB1600	CALL 00DDE741	[Arg1 = 00000011 edocpdfp.00E1E741
00C72B83	. 399E AC000000	CMP DWORD PTR DS:[ESI+AC],EBX	
00C72B89	76 6D	JBE SHORT 00C72BF8	From here
00C72B8B	. 385D F3	CMP BYTE PTR SS:[EBP-D],BL	
00C72B8E	74 68	JE SHORT 00C72BF8	From here
00C72B90	. 8B86 C4000000	MOV EAX,DWORD PTR DS:[ESI+C4]	
00C72B96	. 3D 88000000	CMP EAX,88	
00C72B9B	74 0E	JE SHORT 00C72BAB	
00C72B9D	. 3D 80160000	CMP EAX,1680	
00C72BA2	75 54	JNZ SHORT 00C72BF8	
00C72BA4	. 68 FD000000	PUSH 0FD	
00C72BA9	EB 45	JMP SHORT 00C72BF0	From here

The call at 0C72B40 is our wonder call. So let go to the beginning of this procedure and find out what happens.
Do not trace into the call at 0C72B40 yet.

1.4 ANALYZING THE ALGORITHM

Once we find out where we are directed we end up here:

00C72AD4	. 8BCF	MOV ECX,EDI	
00C72AD6	. E8 76541600	CALL 000D7F51	Get Navn
00C72ADB	. 8B3F	MOV EDI,DWORD PTR DS:[EDI]	
00C72ADD	. 8D4D C4	LEA ECX,DWORD PTR SS:[EBP-3C]	
00C72AE0	. FF77 F8	PUSH DWORD PTR DS:[EDI-8]	
00C72AE3	. 57	PUSH EDI	
00C72AE4	. E8 51CBFEFF	CALL 00C5F63A	Name to Unicode
00C72AE9	. 8B86 A0000000	MOV EAX,DWORD PTR DS:[ESI+A0]	
00C72AEF	. 8D4D A4	LEA ECX,DWORD PTR SS:[EBP-5C]	
00C72AF2	. FF70 F8	PUSH DWORD PTR DS:[EAX-8]	
00C72AF5	. 50	PUSH EAX	
00C72AF6	. E8 3FCBFEFF	CALL 00C5F63A	Companyname to Unicode
00C72AFB	. 8B86 98000000	MOV EAX,DWORD PTR DS:[ESI+98]	
00C72B01	. 8D4D D4	LEA ECX,DWORD PTR SS:[EBP-2C]	
00C72B04	. FF70 F8	PUSH DWORD PTR DS:[EAX-8]	
00C72B07	. 50	PUSH EAX	
00C72B08	. E8 2DCBFEFF	CALL 00C5F63A	Serial to Unicode
00C72B0D	. 399E AC000000	CMP DWORD PTR DS:[ESI+AC],EBX	
00C72B13	✓ 76 35	JBE SHORT 00C72B4A	
00C72B15	. 8A86 C8000000	MOV AL,BYTE PTR DS:[ESI+C8]	
00C72B18	. 50	PUSH EAX	
00C72B1C	. 8D45 B4	LEA EAX,DWORD PTR SS:[EBP-4C]	
00C72B1F	. FFB6 C4000000	PUSH DWORD PTR DS:[ESI+C4]	
00C72B25	. FFB6 C0000000	PUSH DWORD PTR DS:[ESI+C0]	
00C72B2B	. FFB6 BC000000	PUSH DWORD PTR DS:[ESI+BC]	
00C72B31	. FFB6 B8000000	PUSH DWORD PTR DS:[ESI+B8]	
00C72B37	. 50	PUSH EAX	
00C72B38	. 8D45 D4	LEA EAX,DWORD PTR SS:[EBP-2C]	
00C72B3B	. 50	PUSH EAX	
00C72B3C	. 8D45 C4	LEA EAX,DWORD PTR SS:[EBP-3C]	
00C72B3F	. 50	PUSH EAX	
00C72B40	. E8 EEEBFEFF	CALL 00C61733	Let the fun begin
00C72B45	. 83C4 20	ADD ESP,20	
00C72B48	✓ EB 14	JMP SHORT 00C72B5E	
00C72B4A	. 8D45 B4	LEA EAX,DWORD PTR SS:[EBP-4C]	
00C72B4D	. 50	PUSH EAX	
00C72B4E	. 8D45 D4	LEA EAX,DWORD PTR SS:[EBP-2C]	
00C72B51	. 50	PUSH EAX	
00C72B52	. 8D45 C4	LEA EAX,DWORD PTR SS:[EBP-3C]	
00C72B55	. 50	PUSH EAX	
00C72B56	. E8 7EE8FEFF	CALL 00C613D9	
00C72B5B	. 83C4 0C	ADD ESP,0C	
00C72B5E	> 3AC3	CMP AL,BL	
00C72B60	✓ 75 50	JNZ SHORT 00C72BB2	

Some of the comments are just my own comments added while I debugged the target. I will not go into details. You can trace them yourself.

The interesting part is the CALL at 0C72B40 and the CMP at 0C72B5E. If you trace over the call at 0C72B40 you will see that the call returns FALSE. This returned BOOL is then checked at 0C72B5E. So.... The call at 0C72B40 has to return TRUE in order to on.

Tracing into the call at 0C72B40 we first end up here:

00C617A7	. 8B7D 0C	MOV EDI,DWORD PTR SS:[EBP+C]
00C617AA	. C645 FC 04	MOV BYTE PTR SS:[EBP-4],4
00C617AE	. 837F 08 40	CMP DWORD PTR DS:[EDI+8],40
00C617B2	✓ 73 07	JNB SHORT 00C617BB
00C617B4	. 32DB	XOR BL,BL
00C617B6	✓ E9 FA020000	JMP 00C61AB5
00C617BB	> 6A 20	PUSH 20

Take a look at 0C617AE. This compares the length of our name-string with 40h (64 bytes). So we can conclude that our name-string has to be at least 64 bytes long. Set a breakpoint at 0C617AE and press F9. Reenter a 64 byte long name-string. I recommend you type in

00C6190B	. 8D45 CC	LEA EAX,DWORD PTR SS:[EBP-34]	Name-string
00C6190E	. 50	PUSH EAX	
00C6190F	. 8D45 BC	LEA EAX,DWORD PTR SS:[EBP-44]	
00C61912	. 50	PUSH EAX	
00C61913	. 8D45 8C	LEA EAX,DWORD PTR SS:[EBP-74]	
00C61916	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00C61919	. 50	PUSH EAX	
00C6191A	. E8 DBF1FFFF	CALL 00C60AFA	

```

00C60BB9 > 8B45 0C MOV EAX,DWORD PTR SS:[EBP+0C]
00C60BBC B9 00010000 MOV ECX,100
00C60BC1 0FB700 MOVZX EAX,WORD PTR DS:[EAX]
00C60BC4 99 CDQ
00C60BC5 F7F9 IDIV ECX
00C60BC7 8B45 F0 MOV EAX,DWORD PTR SS:[EBP-10]
00C60BCA FEC2 INC DL
00C60BCC 47 INC EDI
00C60BCD 8810 MOV BYTE PTR DS:[EAX],DL
00C60BCF 3B7D E4 CMP EDI,DWORD PTR SS:[EBP-1C]
00C60BD2 72 9F JB SHORT 00C60B73

```

00C60BFF	> 52	PUSH EDI	Last 32 bytes of typed serial
00C60C00	. 53	PUSH EBX	NULL
00C60C01	. 51	PUSH ECX	Last 32 bytes of typed serial
00C60C02	. 50	PUSH EAX	Name-string
00C60C03	. 8D45 9C	LEA EAX, 0WORD PTR SS:[EBP-64]	
00C60C06	. 50	PUSH EAX	
00C60C07	. E8 30280900	CALL 00CF343C	

Let's trace into 0CF343C. It will turn out to be important.

00CF3495	> 6A 20	PUSH 20	
00CF3497	. 50	PUSH EAX	
00CF3498	. 8D85 74FFFFFF	LEA EAX,DWORD PTR SS:[EBP-8C]	
00CF349E	. 50	PUSH EAX	
00CF349F	. E8 BC770000	CALL 00CFAC60	Get ciphered name and rest of subkey
00CF34A4	. 6A 20	PUSH 20	
00CF34A6	. 8D85 74FFFFFF	LEA EAX,DWORD PTR SS:[EBP-8C]	
00CF34AC	. FF75 10	PUSH DWORD PTR SS:[EBP+10]	Last 32 bytes of typed serial
00CF34AF	. 50	PUSH EAX	
00CF34B0	. E8 AB770000	CALL 00CFAC60	Append last 32 bytes and do MD5 hash
00CF34B5	. 8D45 14	LEA EAX,DWORD PTR SS:[EBP+14]	
00CF34B8	. 6A 04	PUSH 4	
00CF34BA	. 50	PUSH EAX	
00CF34BB	. 8D85 74FFFFFF	LEA EAX,DWORD PTR SS:[EBP-8C]	
00CF34C1	. 50	PUSH EAX	
00CF34C2	. E8 99770000	CALL 00CFAC60	Place NULL at start of string
00CF34C7	. 6A 10	PUSH 10	
00CF34C9	. 8D85 74FFFFFF	LEA EAX,DWORD PTR SS:[EBP-8C]	
00CF34CF	. FF75 18	PUSH DWORD PTR SS:[EBP+18]	
00CF34D2	. 50	PUSH EAX	
00CF34D3	. E8 8B770000	CALL 00CFAC60	Place byte 32-48 of typed serial in string
00CF34D8	. 8D85 74FFFFFF	LEA EAX,DWORD PTR SS:[EBP-8C]	
00CF34DE	. 50	PUSH EAX	
00CF34DF	. 8D45 D0	LEA EAX,DWORD PTR SS:[EBP-30]	
00CF34E2	. 50	PUSH EAX	
00CF34E3	. E8 187A0000	CALL 00CFAF00	
00CF34E8	. 6A 32	PUSH 32	Number of times to loop
00CF34EA	. 8D45 D0	LEA EAX,DWORD PTR SS:[EBP-30]	
00CF34ED	. 6A 10	PUSH 10	
00CF34EF	. 50	PUSH EAX	MD5 digest
00CF34F0	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00CF34F3	. E8 23000000	CALL 00CF351B	Do MD5 hash loop

Below I will try to describe what happens.

Address	Hex dump	ASCII
0012E158	33 6C 42 45 28 BF 4E 5E 4E 75 8A 41 64 00 4E 56	31BE(1N^NuêAd.NU
0012E168	FF FA 01 08 2E 2E 00 B6 D0 68 3E 80 2F 0C A9 FE	·0...A&h>C/.0■
0012E178	00 00 00 00 80 2C C6 00 01 00 00 00 04 00 00 00	C.s.0...♦...
0012E188	34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34	4444444444444444

The call at 0CF34B0 append last 32 bytes to name-string and then do a MD5 hash on the string.

Address	Hex dump	ASCII
0012E158	33 6C 42 45 28 BF 4E 5E 4E 75 8A 41 64 00 4E 56	31BE(1N^NuêAd.NU
0012E168	FF FA 01 08 2E 2E 00 B6 D0 68 3E 80 2F 0C A9 FE	·0...A&h>C/.0■
0012E178	33 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33	3333333333333333
0012E188	34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34	4444444444444444
0012E198	20 00 00 00 4C F2 12 00 2C 86 DF 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Here the MD5 hash is performed:

00CFAD3B	> 6A 01	PUSH 1	
00CFAD3D	. 55	PUSH EBP	Pointer to string
00CFAD3E	. 53	PUSH EBX	MD5 digest
00CFAD3F	. E8 2CF8FFFF	CALL 00CFA570	MD5 hash

This is how the digest looks after the MD5 hash:

Address	Hex dump	ASCII
0012E140	5C 07 9F 4E A1 F7 88 A0 2B 80 E2 C1 23 FA 14 6F	\.fNi-ã+Çö-#·¶o
0012E150	00 02 00 00 00 00 00 00 33 6C 42 45 28 BF 4E 5E	.0.....3lBE(γH^
0012E160	4E 75 8A 41 64 00 4E 56 FF FA 01 08 2E 2E 00 B6	NuëAd.NU ·6...ã
0012E170	D0 68 3E 80 2F 0C A9 FE 33 33 33 33 33 33 33	sh>Ç/.@■33333333
0012E180	33 33 33 33 33 33 33 33 34 34 34 34 34 34 34	3333333334444444
0012E190	34 34 34 34 34 34 34 34 20 00 00 00 4C E2 12 00	44444444 ...L0+
0012E198	7C 86 DF 00 00 00 00 00 00 00 00 00 00 00 00	!B? ▲...+*

Calls at 0CF34C2 and 0CF34D3 creates a string that looks like this:

Address	Hex dump	ASCII
0012E158	00 00 00 00 33 33 33 33 33 33 33 33 33 33 33	333333333333
0012E168	33 33 33 33 00 00 00 00 00 00 00 00 00 00 00	3333Ç.....
0012E178	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ã0.....
0012E188	00 00 00 00 00 00 00 00 A0 02 00 00 00 00 00	ã0.....
0012E198	14 00 00 00 4C E2 12 00 7C 86 DF 00 00 00 00 00	¶...L0+!B?

Again a MD5 hash is performed, resulting in this digest:

Address	Hex dump	ASCII
0012E140	29 89 B1 62 A4 75 BA D8 DD 30 A4 92 6F BC 0B 0A	)ëBkull i!0R#co?d.
0012E150	A0 02 00 00 00 00 00 00 00 00 00 00 33 33 33	ã0.....3333
0012E160	33 33 33 33 33 33 33 33 33 33 33 33 00 00 00	333333333333Ç...
0012E170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0012E180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0012E190	A0 02 00 00 00 00 00 00 14 00 00 00 4C E2 12 00	ã0.....¶...L0+.

Notice that the calls doesn't initialize MD5 before hashing this part !!

The call, at 0CF34F3, does a MD5 hash loop. This time they do initialize MD5 before hashing. The loop looks like this:

```

00CF3545 | . 50          PUSH EAX
00CF3546 | . 8975 FC     MOV DWORD PTR SS:[EBP-4],ESI
00CF3549 | . FF75 10     PUSH DWORD PTR SS:[EBP+10]
00CF354C | . FF75 0C     PUSH DWORD PTR SS:[EBP+C]
00CF354F | . E8 EC7A0000 CALL 00CFB040
00CF3554 | . 8B45 14     MOV EAX,DWORD PTR SS:[EBP+14]
00CF3557 | . 83C4 0C     ADD ESP,0C
00CF355A | . 3BC6       CMP EAX,ESI
00CF355C | . 7E 18     JLE SHORT 00CF3576
00CF355E | . 8D78 FF     LEA EDI,DWORD PTR DS:[EAX-1]
00CF3561 | > 8D45 D0     LEA EAX,DWORD PTR SS:[EBP-30]
00CF3564 | . 50          PUSH EAX
00CF3565 | . 8D45 D0     LEA EAX,DWORD PTR SS:[EBP-30]
00CF3568 | . 6A 10     PUSH 10
00CF356A | . 50          PUSH EAX
00CF356B | . E8 D07A0000 CALL 00CFB040
00CF3570 | . 83C4 0C     ADD ESP,0C
00CF3573 | . 4F         DEC EDI
00CF3574 | . ^ 75 EB     JNZ SHORT 00CF3561
00CF3576 | > 8D45 D0     LEA EAX,DWORD PTR SS:[EBP-30]

```

The call at OCF354F and OCF356B does both init and hashing. The result of the loop is this:

Address	Hex dump	ASCII
0012E0BC	A5 09 6C B4 20 27 85 7A 3D 8E 94 B2 E8 05 A4 F3	A..H '32=A0...n%
0012E0CC	00 2C C6 00 00 00 00 00 00 00 00 00 00 00 00	..8.....
0012E0DC	00 00 00 00 C0 E1 12 00 9B 89 DF 00 01 00 00 00	...Lp+.e=.0...
0012E0EC	CC E1 12 00 E8 34 CE 00 F4 F1 12 00 9C F1 12 00	...84B .e+.e+

This key is very important. It's used for validating version of the license. I will describe this later.

Once we return with a fresh MD5 digest we can continue tracing again and further down you will see this:

00C60C4A	> 53	PUSH EBX	NULL
00C60C4B	. 68 80000000	PUSH 80	
00C60C50	. 6A 10	PUSH 10	
00C60C52	. 56	PUSH ESI	MD5 Digest
00C60C53	. 6A 10	PUSH 10	
00C60C55	. 50	PUSH EAX	
00C60C56	. 8D45 9C	LEA EAX,DWORD PTR SS:[EBP-64]	First 32 bytes of typed serial (hex)
00C60C59	. 50	PUSH EAX	
00C60C5A	. E8 DE360900	CALL 00CF433D	
00C60C5F	. 83C4 1C	ADD ESP,1C	
00C60C60	. 5505 50000000	PUSH DWORD PTR DS:[EIP+5]	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Lets trace into OCF433D, and eventually we end up here:

00CF4379	. 8D45 F0	LEA EAX,DWORD PTR SS:[EBP-10]	
00CF437C	. FF75 1C	PUSH DWORD PTR SS:[EBP+1C]	
00CF437F	. FF75 18	PUSH DWORD PTR SS:[EBP+18]	
00CF4382	. FF75 14	PUSH DWORD PTR SS:[EBP+14]	MD5 digest (key)
00CF4385	. 50	PUSH EAX	
00CF4386	. E8 E4FEFFFF	CALL 00CF426F	Create magic table
00CF438B	. FF75 10	PUSH DWORD PTR SS:[EBP+10]	

Again we trace into OCF426F. There are 2 important calls we have to be aware of. The first is here:

00CF42A2	. FF75 18	PUSH DWORD PTR SS:[EBP+18]	
00CF42A5	. 8D45 04	LEA EAX,DWORD PTR SS:[EBP-2C]	
00CF42A8	. FF75 14	PUSH DWORD PTR SS:[EBP+14]	
00CF42AB	. FF75 10	PUSH DWORD PTR SS:[EBP+10]	
00CF42AE	. FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
00CF42B1	. 50	PUSH EAX	
00CF42B2	. E8 D3EFFFFF	CALL 00CF328A	Do MD5 hash on key
00CF42B3	. 83C4 14	ADD ESP,14	

This call does MD5 hash on the key.

00CF33BD	. 51	PUSH ECX	
00CF33BE	. 56	PUSH ESI	
00CF33BF	. 50	PUSH EAX	MD5 digest (key)
00CF33C0	. E8 7B7C0000	CALL 00CFB040	Do MD5 hash on key
00CF33C5	. 83C4 0C	ADD ESP,0C	

Address	Hex dump	ASCII
01495E21	C8 D4 9C B4 7D FC 10 9F 4A E2 CE C8 FF C3 AF CE	E6E1 j? J0P# >??
01495E31	F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D	: . .: . .: . .: . .
01495E41	F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D	: . .: . .: . .: . .

00CF42E9	> 50	PUSH EAX	MDS digest
00CF42EA	. FF75 EC	PUSH DWORD PTR SS:[EBP-14]	Size of digest
00CF42ED	. 56	PUSH ESI	Output buffer (magic table)
00CF42EE	. E8 4D730000	CALL 00CFB640	Create magic table

Address	Hex dump	ASCII
0113A220	00 00 00 00 00 00 00 00 C8 00 00 00 90 00 00 00	0.
0113A230	3B 00 00 00 F2 00 00 00 6E 00 00 00 37 00 00 00	...n...?
0113A240	59 00 00 00 47 00 00 00 82 00 00 00 60 00 00 00	Y...G...m...
0113A250	48 00 00 00 18 00 00 00 23 00 00 00 BA 00 00 00	H...t...#... .
0113A260	B0 00 00 00 B3 00 00 00 65 00 00 00 4A 00 00 00	...e...J.
0113A270	80 00 00 00 BF 00 00 00 2E 00 00 00 61 00 00 00	...a...
0113A280	EE 00 00 00 39 00 00 00 92 00 00 00 25 00 00 00	...9...E...%
0113A290	75 00 00 00 6C 00 00 00 C1 00 00 00 9F 00 00 00	u...t...f...
0113A2A0	20 00 00 00 F3 00 00 00 8F 00 00 00 E8 00 00 00	...A...p...
0113A2B0	FC 00 00 00 D1 00 00 00 31 00 00 00 28 00 00 00	...0...1...()
0113A2C0	5E 00 00 00 FB 00 00 00 A8 00 00 00 9E 00 00 00	^...t...x...
0113A2D0	96 00 00 00 89 00 00 00 B4 00 00 00 85 00 00 00	G...e...t...a...
0113A2E0	F9 00 00 00 36 00 00 00 4D 00 00 00 52 00 00 00	...6...M...R...
0113A2F0	14 00 00 00 24 00 00 00 29 00 00 00 49 00 00 00	...\$...).I...
0113A300	07 00 00 00 00 00 00 00 22 00 00 00 4C 00 00 00	...

00CF438B	FF75 10	PUSH DWORD PTR SS:[EBP+10]	
00CF438E	57	PUSH EDI	
00CF439F	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	First 32 bytes of typed serial (hex)
00CF4392	FF75 F0	PUSH DWORD PTR SS:[EBP-10]	Magic table
00CF4395	E8 7CFFFFFF	CALL 00CF4316	

00CF4316	55	PUSH EBP	
00CF4317	8BEC	MOV EBP,ESP	
00CF4319	8B45 14	MOV EAX,DWORD PTR SS:[EBP+14]	
00CF431C	85C0	TEST EAX,EAX	
00CF431E	75 09	JNZ SHORT 00CF4329	
00CF4320	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
00CF4323	E8 D8E00600	CALL 00D62400	
00CF4328	59	POP ECX	
00CF4329	FF75 10	PUSH DWORD PTR SS:[EBP+10]	
00CF432C	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
00CF432F	50	PUSH EAX	
00CF4330	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00CF4333	E8 586D0000	CALL 00CFB090	
00CF4338	83C4 10	ADD ESP,10	
00CF433B	50	POP EBP	
00CF433C	C3	RETN	

Output
First 32 bytes of typed serial (hex)

Magic table
Get new digest (not MD5)

Output looks like this:

Address	Hex dump	ASCII
01495F08	4E 85 AF 4D 57 98 C4 04 BF 90 86 0B C1 7C 75 24	Nä»Mw»-»é»+!u\$
01495FE8	AB AB AB AB AB AB AB AB EE FE EE FE EE FE EE FE	#####-#####
01495FF8	00 00 00 00 00 00 00 00	

So should think that the worst part was over but no, no. We are not done yet. It seems like the developers created this algorithm just to bore us. But they will not succeed!!

Earlier we followed at call at 0C6191A. We now return to 0C6191F and trace down a little until we stumble upon this:

00C61946	8045 AC	LEA EAX,DWORD PTR SS:[EBP-54]
00C61949	50	PUSH EAX
00C6194A	8045 9C	LEA EAX,DWORD PTR SS:[EBP-64]
00C6194D	50	PUSH EAX
00C6194E	8045 8C	LEA EAX,DWORD PTR SS:[EBP-74]
00C61951	FF75 08	PUSH DWORD PTR SS:[EBP+8]
00C61954	50	PUSH EAX
00C61955	E8 1AEFFFFF	CALL 00C60774

Trace in 0C60774. The beginning of this procedure do the modulo hash on the name once again and stores it in memory. It's going to use this later. Further down we end here:

00C608C3	8045 B0	LEA EAX,DWORD PTR SS:[EBP-50]
00C608C6	50	PUSH EAX
00C608C7	8045 D0	LEA EAX,DWORD PTR SS:[EBP-30]
00C608CA	50	PUSH EAX
00C608CB	8045 8C	LEA EAX,DWORD PTR SS:[EBP-74]
00C608CE	50	PUSH EAX
00C608CF	E8 5C330900	CALL 00CF3C30

The procedure at 0CF3C30 is yet another attempt to bore us... But let trace into it. A little down we find this:

00CF3CD9	> 6A 33	PUSH 33	Number of loops
00CF3CDB	. 6A 20	PUSH 20	Size of string to hash
00CF3CDD	. 50	PUSH EAX	
00CF3CDE	. 8045 B0	LEA EAX, DWORD PTR SS:[EBP-50]	
00CF3CE1	. 50	PUSH EAX	
00CF3CE2	. E8 34F8FFFF	CALL 00CF351B	Do yet another MD5 hash loop

Remember that the new digest looks like this:

Address	Hex dump	ASCII
0012E09C	4E 85 AF 4D 57 98 C4 04 BF 90 86 0B C1 7C 75 24	Nä»MWÜ-~ªã±!u\$
0012E0AC	28 BF 4E 5E 4E 75 8A 41 64 00 4E 56 FF FA 01 08	(7N^NuêAd.NU ·0
0012E0BC	80 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Ç.....

It has to be run through the MD5 loops yet 51 times. The outcome is this:

Address	Hex dump	ASCII
0012E0FC	B7 DA 82 59 11 40 3B 91 45 24 DC BC 5D B9 07 8A	ÄreY@:æE\$~"Ji·ê
0012E10C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0012E11C	00 00 00 00 94 E1 12 00 9B 89 DF 00 01 00 00 00	ô6±.øë~.0...

Back at 0C608D4 we can trace down a little again, and we end up here:

00CF3D1D	> 6A 20	PUSH 20	Ciphered name-string Output size MD5 digest
00CF3D1F	. 51	PUSH ECX	
00CF3D20	. FF75 E8	PUSH DWORD PTR SS:[EBP-18]	
00CF3D23	. 50	PUSH EAX	
00CF3D24	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00CF3D27	. E8 9AF8FFFF	CALL 00CF35C6	

I'm really bored now!! If you trace into this call you will see that the MD5 digest is used to build yet another magic table, then xor'ed again. This is done 20 times. Giving an output that looks like this:

Address	Hex dump	ASCII
01495DB8	B8 88 2A 39 17 6D F9 66 CA 60 E3 28 77 47 C6 3A	0ë*9±m-f^'d(wG\$:
01495DC8	27 1B 07 A1 37 84 0D F3 98 85 24 59 3E 3C DA 83	'+.i7ä.%ÿ\$Y><râ
01495DD8	AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00	%%%%%...

Finally this bored task has ended. We are now back at 0C608D4. A little further down we see this:

00C608F9	. 33FF	XOR EDI,EDI	
00C608FB	. 8DB5 68FFFF	LEA ESI,DWORD PTR SS:[EBP-98]	
00C60901	> 397D C8	CMP DWORD PTR SS:[EBP-38],EDI	
00C60904	~ 72 14	JB SHORT 00C6091A	
00C60906	. 395D C4	CMP DWORD PTR SS:[EBP-3C],EBX	
00C60909	~ 74 0F	JE SHORT 00C6091A	
00C6090B	. 8D4D C0	LEA ECX,DWORD PTR SS:[EBP-40]	
00C6090E	. E8 50210000	CALL 00C62A63	
00C60913	. 8B45 C4	MOV EAX,DWORD PTR SS:[EBP-3C]	
00C60916	. 03C7	ADD EAX,EDI	
00C60918	~ EB 05	JMP SHORT 00C6091F	
00C6091A	> B8 0C99E000	MOV EAX,00E0990C	
00C6091F	> 0FB600	MOVZX EAX,BYTE PTR DS:[EAX]	
00C60922	. 50	PUSH EAX	
00C60923	. 68 141BE600	PUSH 00E61B14	ASCII "%02x"
00C60928	. 56	PUSH ESI	
00C60929	. E8 1C381000	CALL 00D6414A	
00C6092E	. 83C4 0C	ADD ESP,0C	
00C60931	. 47	INC EDI	
00C60932	. 46	INC ESI	
00C60933	. 46	INC ESI	
00C60934	. 83FF 10	CMP EDI,10	
00C60937	^ 7C C8	JL SHORT 00C60901	

There are two such loops. The first one (the above) turns our latest key into a text-string, giving us this output:

Address	Hex dump	ASCII
0012E1C0	62 38 38 38 32 61 33 39 31 37 36 64 66 39 36 36	b8882a39176df966
0012E1D0	63 61 36 30 65 33 32 38 37 37 34 37 63 36 33 61	ca60e3287747c63a
0012E1E0	00 0C C6 00 00 00 00 00 00 00 00 00 00 00 00	...a.....

The second loop is not important, or at least we don't need it!

Let's back trace until we return at 0C6195A. Above we see a byte-to-byte comparison loop of the key above with the last 32 bytes typed in in our serial.

00C61967	> 3975 D4	CMP DWORD PTR SS:[EBP-2C],ESI	
00C6196A	~ 72 16	JB SHORT 00C61982	
00C6196C	. 837D D0 00	CMP DWORD PTR SS:[EBP-30],0	
00C61970	~ 74 10	JE SHORT 00C61982	
00C61972	. 8D4D CC	LEA ECX,DWORD PTR SS:[EBP-34]	
00C61975	. E8 E9100000	CALL 00C62A63	
00C6197A	. 8B45 D0	MOV EAX,DWORD PTR SS:[EBP-30]	
00C6197D	. 8D3C30	LEA EDI,DWORD PTR DS:[EAX+ESI]	
00C61980	~ EB 05	JMP SHORT 00C61987	
00C61982	> BF 0C99E000	MOV EDI,00E0990C	
00C61987	> 3975 B4	CMP DWORD PTR SS:[EBP-4C],ESI	
00C6198A	~ 72 1A	JB SHORT 00C619A6	
00C6198C	. 837D B0 00	CMP DWORD PTR SS:[EBP-50],0	
00C61990	~ 74 14	JE SHORT 00C619A6	
00C61992	. 8D4D AC	LEA ECX,DWORD PTR SS:[EBP-54]	
00C61995	. E8 EC29FEFF	CALL 00C44386	
00C6199A	. 8B45 B0	MOV EAX,DWORD PTR SS:[EBP-50]	
00C6199D	. BB EC88E000	MOV EBX,00E088EC	
00C619A2	. 03C6	ADD EAX,ESI	
00C619A4	~ EB 07	JMP SHORT 00C619AD	
00C619A6	> BB EC88E000	MOV EBX,00E088EC	
00C619AB	. 8BC3	MOV EAX,EBX	
00C619AD	> 8A0F	MOV CL,BYTE PTR DS:[EDI]	byte-to-byte check
00C619AF	. 3A08	CMP CL,BYTE PTR DS:[EAX]	
00C619B1	~ 75 0D	JNZ SHORT 00C619C0	
00C619B3	. 46	INC ESI	
00C619B4	. 83FE 20	CMP ESI,20	
00C619B7	^ 7C AE	JL SHORT 00C61967	

```
Registers (FPU)
EAX 0113A221 ASCII "b8882a39176df966ca60e3287747c63a"
ECX 0012E29C
EDX 01130608
EBX 00E088EC edocpdfp.00E088EC
ESP 0012E270
EBP 0012E2F0
ESI 00000000
EDI 01495C79 ASCII "33333333333333334444444444444444"
EIP 00C619AD edocpdfp.00C619AD
```

It's now clear that our last part of the serial has to be the one calculated in what I call the magic table. So let's sum up what we know:

The last 32 bytes of the serial is being calculated using a first key, name, MD5 hashing and the magic table. This is being checked. So if the output of these calculations are not equal to last 32 bytes of serial we get a "bad boy"

The first key is being calculated from the last 32 bytes of serial using our name and MD5 hashing.

Now... My mind was confused!! How can we calculate any of these since they are both dependent on each other? It's not possible.... Or is it? Brute-forcing is out of the question since the key is 128 bit, and creating a MD5 collision seems pretty far out... What to do?

After a little thinking it became clear!! The key to this is the magic table and the first key. I will show how it's done later.

1.5 FIGURING OUT A VALID KEY

We know the target does a byte-to-byte comparison on the last 32 bytes of the serial. If this is not a success, the target performs a AND and creates a bad-boy situation:

00C619AB	. 8BC3	MOV EAX,EBX	byte-to-byte check
00C619AD	> 8A0F	MOV CL,BYTE PTR DS:[EDI]	
00C619AF	. 3A08	CMP CL,BYTE PTR DS:[EAX]	
00C619B1	~ 75 0D	JNZ SHORT 00C619C0	
00C619B3	. 46	INC ESI	Create BAD-BOY
00C619B4	. 83FE 20	CMP ESI,20	
00C619B7	^ 7C AE	JL SHORT 00C61967	
00C619B9	. BB EC88E000	MOV EBX,00E088EC	
00C619BE	~ EB 04	JMP SHORT 00C619C4	
00C619C0	> 8065 F3 00	AND BYTE PTR SS:[EBP-D],0	

In order to get a valid key we need to look at quite a few checks. In my opinion this is easier in IDA using the graphical view. Therefore the next part will be using part Olly, part IDA.

```

00C619E2 | . 8BC3      MOV EAX,EBX
00C619E4 | > 50        PUSH EAX
00C619E5 | . 57        PUSH EDI
00C619E6 | . E8 5CF9FFF CALL 00C61347
00C619E8 | . 8945 08    MOV DWORD PTR SS:[EBP+8],EAX
00C619EE | . 59        POP ECX
00C619EF | . 66:C1E8 0D SHR AX,0D
00C619F3 | . 66:3D 0100 CMP AX,1
00C619F7 | . 59        POP ECX
00C619F8 | . 75 03     JNZ SHORT 00C619FD
00C619FA | . 8845 F2    MOV BYTE PTR SS:[EBP-E],AL
    
```

The above is taking the key[0] (word size) and bit shifting 13 bits right. If the 13th bit is set, then we are running a test-version, if not set we're a real application. I don't want to go too much into detail about this. Feel free to test it yourself. If 13th bit is set the target goes on to find out how many days are left of trial, using key [6] (word) to find out.

```

00C61A0B | . 8BC3      MOV EAX,EBX
00C61A0D | > 50        PUSH EAX
00C61A0E | . 6A 04     PUSH 4
00C61A10 | . E8 32F9FFF CALL 00C61347
00C61A15 | . 807D F2 00 CMP BYTE PTR SS:[EBP-E],0
00C61A19 | . 59        POP ECX
00C61A1A | . 59        POP ECX
00C61A1B | . 74 11     JE SHORT 00C61A2E
00C61A1D | . 57        PUSH EDI
00C61A1E | . 50        PUSH EAX
00C61A1F | . E8 28CD0600 CALL 00CCE74C
00C61A24 | . 59        POP ECX
00C61A25 | . 84C0     TEST AL,AL
00C61A27 | . 59        POP ECX
00C61A28 | . 74 04     JE SHORT 00C61A2E
00C61A2A | . 8065 F3 00 AND BYTE PTR SS:[EBP-D],0
    
```

Are we a test-version ?

As one might can see, EBP-0E is check if FALSE. If so, we go on, else we test for how many days left of trial. If none left, create BAD BOY at 0C61A2A.

Next we check key[4](word):

```

00C61A35 | . 8BC3      MOV EAX,EBX
00C61A37 | > 50        PUSH EAX
00C61A38 | . 6A 02     PUSH 2
00C61A3A | . E8 08F9FFF CALL 00C61347
00C61A3F | . 59        POP ECX
00C61A40 | . 59        POP ECX
00C61A41 | . 33C9     XOR ECX,ECX
00C61A43 | . 8ACC     MOV CL,AH
00C61A45 | . 25 FF000000 AND EAX,0FF
00C61A4A | . 394D 1C   CMP DWORD PTR SS:[EBP+1C],ECX
00C61A4D | . 74 0D     JE SHORT 00C61A5C
00C61A4F | . 3BC7     CMP EAX,EDI
00C61A51 | . 74 05     JE SHORT 00C61A58
00C61A53 | . 3945 1C   CMP DWORD PTR SS:[EBP+1C],EAX
00C61A56 | . 76 04     JBE SHORT 00C61A5C
00C61A58 | > 8065 F3 00 AND BYTE PTR SS:[EBP-D],0
    
```

CMP ecx == 5

CMP eax >= 5

Assume the key is this: **xxxxYYZZxxxxxxxxxxxxxxxxxxxxxxxxxxxx**

From the above we can see the either ZZ has to be 05 or YY has to be above 05.

Next we encounter this:

```

00C61A63 . 8BC3      MOV EAX,EBX
00C61A65 > 50        PUSH EAX
00C61A66 . 6A 0C     PUSH 0C
00C61A68 . E8 DAF8FFFF CALL 00C61347
00C61A6D . 3945 18   CMP DWORD PTR SS:[EBP+18],EAX    CMP eax == 5900
00C61A70 . 59        POP ECX
00C61A71 . 59        POP ECX
00C61A72 > 74 04     JE SHORT 00C61A78
00C61A74 . 8065 F3 00 AND BYTE PTR SS:[EBP-D],0
00C61A78 . 8075 00   AND BYTE PTR SS:[EBP+00],0

```

Key[12] (word) has to be equal to 0x5900. So now we know a key so far could look like this:

xxxx1205xxxxxxxxxxxxxxxx0059xxxx

Last but not least we see this:

```

00C61A84 . 8BC3      MOV EAX,EBX
00C61A86 > 50        PUSH EAX
00C61A87 . 6A 0F     PUSH 0F
00C61A89 . E8 B9F8FFFF CALL 00C61347
00C61A8E . 59        POP ECX
00C61A8F . 59        POP ECX
00C61A90 . 0FB74D 08 MOVZX ECX,WORD PTR SS:[EBP+8]    Get key[0] (word) && 0x1FFF
00C61A94 . 394D 14   CMP DWORD PTR SS:[EBP+14],ECX
00C61A97 > 74 05     JE SHORT 00C61A9E
00C61A99 . 394D 20   CMP DWORD PTR SS:[EBP+20],ECX
00C61A9C > EB 0E     JMP SHORT 00C61AAC
00C61A9E > 8445 24   TEST BYTE PTR SS:[EBP+24],AL    Test AL = 0x80
00C61AA1 > 75 0F     JNZ SHORT 00C61AB2
00C61AA3 > EB 09     JMP SHORT 00C61AAE
00C61AA5 > 0FB745 08 MOVZX EAX,WORD PTR SS:[EBP+8]
00C61AA9 . 3945 14   CMP DWORD PTR SS:[EBP+14],EAX
00C61AAC > 74 04     JE SHORT 00C61AB2
00C61AAE > 8065 F3 00 AND BYTE PTR SS:[EBP-D],0
00C61AB2 > 8A5D F3   MOV BL,BYTE PTR SS:[EBP-D]    Move GOOD BOY / BAD BOY to BL
00C61AB4 . 60        PUSH 0

```

First we get key[15](byte). Then we check if key[0](word) is either equal to 0x1680 or 0x1688. If key[0] not equal to any of these values, BAD BOY is created. The same goes for key[15], if not the 8th bit is set. Earlier in the code key[0] is AND'ed with 0x1FFF .

Right now we have a key that could look like this:

88161205xxxxxxxxxxxxxxxx0059xx80

Unfortunately we are not done yet. We still need to fill in a few gabs. To find the missing pieces a good idea would be to set a breakpoint on the byte-to-byte comparison. Close the registration window and press the question mark. You will now be in the byte-to-byte comparison again, now fake your way through the key validation, and you will end up here:

00C61E56	. 83C4 24	ADD ESP,24	
00C61E59	. 3AC3	CMP AL,BL	
00C61E5B	. 8845 F3	MOV BYTE PTR SS:[EBP-D],AL	
00C61E5E	√ 74 09	JE SHORT 00C61E69	
00C61E60	. 395D E8	CMP DWORD PTR SS:[EBP-18],EBX	
00C61E63	√ 0F85 3A010000	JNZ 00C61FA3	
00C61E69	> 53	PUSH EBX	
00C61E6A	. FF75 18	PUSH DWORD PTR SS:[EBP+18]	
00C61E6D	. FF75 14	PUSH DWORD PTR SS:[EBP+14]	
00C61E70	. FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
00C61E73	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00C61E76	. E8 A5740600	CALL 00CC9320	
00C61E7B	. 8A45 0F	MOV AL,BYTE PTR SS:[EBP+F]	
00C61E7E	. 83C4 14	ADD ESP,14	
00C61E81	. 8D4D C0	LEA ECX,DWORD PTR SS:[EBP-40]	
00C61E84	. 8845 C0	MOV BYTE PTR SS:[EBP-40],AL	
00C61E87	. 53	PUSH EBX	
00C61E88	. E8 CDC7FEFF	CALL 00C4E65A	
00C61E8D	. 68 041BE600	PUSH 00E61B04	UNICODE "UserID"
00C61E92	. E8 15181000	CALL 00D636AC	
00C61E97	. 59	POP ECX	
00C61E98	. 50	PUSH EAX	
00C61E99	. 68 041BE600	PUSH 00E61B04	UNICODE "UserID"
00C61E9E	. 8D4D C0	LEA ECX,DWORD PTR SS:[EBP-40]	
00C61EA1	. E8 A9CFEFFF	CALL 00C4EE4F	

Hit shift+F9 and you end here:

00C92499	. 83C4 28	ADD ESP,28	
00C9249C	. 84C0	TEST AL,AL	
00C9249E	√ 75 3E	JNZ SHORT 00C924DE	
00C924A0	. 8A86 64010000	MOV AL,BYTE PTR DS:[ESI+164]	
00C924A6	. 50	PUSH EAX	
00C924A7	. 8D45 CC	LEA EAX,DWORD PTR SS:[EBP-34]	
00C924AA	. FFB6 60010000	PUSH DWORD PTR DS:[ESI+160]	
00C924B0	. FFB6 5C010000	PUSH DWORD PTR DS:[ESI+15C]	
00C924B6	. FFB6 58010000	PUSH DWORD PTR DS:[ESI+158]	
00C924BC	. FFB6 54010000	PUSH DWORD PTR DS:[ESI+154]	
00C924C2	. 53	PUSH EBX	
00C924C3	. 68 1810E600	PUSH 00E61018	UNICODE "Software\ITEKSOFT\eDocPrinter\PDF\5.0
00C924C8	. 50	PUSH EAX	
00C924C9	. 8D45 BC	LEA EAX,DWORD PTR SS:[EBP-44]	
00C924CC	. 50	PUSH EAX	
00C924CD	. 57	PUSH EDI	
00C924CE	. E8 DAF7FCFF	CALL 00C61CAD	
00C924D3	. 83C4 28	ADD ESP,28	
00C924D6	. 84C0	TEST AL,AL	
00C924D8	√ 0F84 74060000	JE 00C92B52	
00C924DE	> 8A45 F2	MOV AL,BYTE PTR SS:[EBP-E]	
00C924E1	. 6A 00	PUSH 0	
00C924E3	. 8D4D 9C	LEA ECX,DWORD PTR SS:[EBP-64]	
00C924E6	. 8845 9C	MOV BYTE PTR SS:[EBP-64],AL	
00C924E9	. E8 6CC1FBFF	CALL 00C4E65A	
00C924EE	. BF 041BE600	MOV EDI,00E61B04	UNICODE "UserID"
00C924F3	. 57	PUSH EDI	
00C924F4	. E8 B3110D00	CALL 00D636AC	
00C924F9	. 59	POP ECX	
00C924FD	. 50	PUSH EAX	

Scroll down a little and you will see this:

00C9261E	. 8BCB	MOV ECX,EBX	
00C92620	. E8 D3701400	CALL 00D096F8	
00C92625	. 6A 20	PUSH 20	
00C92627	. 6A 20	PUSH 20	
00C92629	. 8BCB	MOV ECX,EBX	
00C9262B	. E8 20531400	CALL 00D07950	
00C92630	. 8BCB	MOV ECX,EBX	
00C92632	. E8 30741400	CALL 00D09A67	
00C92637	. 8B45 D0	MOV EAX,DWORD PTR SS:[EBP-30]	
00C9263A	. BF EC88E000	MOV EDI,00E088EC	
00C9263F	. 85C0	TEST EAX,EAX	
00C92641	. 75 02	JNZ SHORT 00C92645	
00C92643	. 8BC7	MOV EAX,EDI	
00C92645	. 50	PUSH EAX	Get version from key
00C92646	. 6A 06	PUSH 6	
00C92648	. E8 FAECFCFF	CALL 00C61347	
00C9264D	. 8945 DC	MOV DWORD PTR SS:[EBP-24],EAX	
00C92650	. 8B45 D0	MOV EAX,DWORD PTR SS:[EBP-30]	
00C92653	. 59	POP ECX	
00C92654	. 85C0	TEST EAX,EAX	

Arg2 = 00000020
Arg1 = 00000020
edocpdfp.00E17950

Here the target begins the process of testing the key again. The call at 0C92648 get key[6](word) and stores it. This value is used later for find out what license the target intended for.

A little ways down we stumble upon this case-switch:

00C926D8	. 8B45 DC	MOV EAX,DWORD PTR SS:[EBP-24]	
00C926DE	. 05 CECFFFFF	ADD EAX,-3032	
00C926E3	. 83F8 0F	CMP EAX,0F	
00C926E6	. 0F87 55030000	JA 00C92A41	
00C926EC	. FF2485 C42BC	JMP DWORD PTR DS:[EAX*4+C92BC4]	
00C926F3	. 8DBE F4000000	LEA EDI,DWORD PTR DS:[ESI+F4]	
00C926F9	. 68 CC000000	PUSH 0CC	
00C926FE	. 8BCF	MOV ECX,EDI	
00C92700	. E8 3CC01400	CALL 00DDE741	
00C92705	. 68 CD000000	PUSH 0CD	
00C9270A	. 8D4D EC	LEA ECX,DWORD PTR SS:[EBP-14]	
00C9270D	. E8 2FC01400	CALL 00DDE741	
00C92712	. 8D45 EC	LEA EAX,DWORD PTR SS:[EBP-14]	
00C92715	. 50	PUSH EAX	
00C92716	. 8D45 DC	LEA EAX,DWORD PTR SS:[EBP-24]	
00C92719	. 68 2CFDE600	PUSH 00E6FD9C	
00C9271E	. 50	PUSH EAX	
00C9271F	. E8 56711400	CALL 00D0987A	

Switch (cases 3032..3041)
Case 3032 of switch 00C926DE
Arg1 = 000000CC
edocpdfp.00E1E741
Arg1 = 000000CD
edocpdfp.00E1E741
UNICODE " 1 "

To make a long story short. There are 9 different cases. Each is a different license type. They are listed as followed:

- 3032 - Workstation - 1 license
- 3033 - Workstation - xxx licenses
- 3037 - Terminal Server License - xxx licenses
- 3041 - Terminal Server Administrator Mode License - xxx licenses
- 303E - Terminal Server Connection License - xxx licenses
- 303B - Intranet-server License - xxx licenses
- 303C - Internet Report Server License - xxx licenses
- 303D - Internet Application Service Provider License - xxx licenses
- 3039 - Internet Server License - xxx licenses

As one can see, some of the licences-types takes number of licences. The key also provide that information. Key[8] is the number of licenses. This number has to be greater than 1 and less than 60001. Just trace into one of the cases to check this.

So to create a valid key for a Workstation with 3 licenses the key must look like this:

88161205xxxx33300300xxxx0059xx80

x = random numbers.

Notice that the last byte (key[15]) has to be 0xC0 for advanced edition. Just debug PDFsealer.exe to see for yourself.

1.6 REVERSING THE ALGORITHM

I have included my keygen for this target, but in order to be fair to the software developers I suggest that you **do not** use it. It's only fair that you, the reverser, yourself create a valid serial. If you follow the next steps its possible...

As told earlier I could quite understand how I was to reverse this since the key and last part of the serial was depending on each other.. Well, here is how I did it:

1. Run the magic table trick as described above, using the key (**88161205xxxx33300300xxxx0059xx80**) as input instead of calculated key.
2. We now have the last 32 bytes of out serial, append them to the name-string (the modulo'ed name with appended data). Run the MD5 loop and the creation of the magic table.
3. Instead of using the calculated key use our key (**88161205xxxx33300300xxxx0059xx80**) and run the xor cipher. This will reveal the first 32 bytes of the serial.
4. We now have a 64 bytes serial to use for the target.

1.7 FINAL WORDS...

This has been a fun task.. When I first started on reversing this protection I thought nothing of it. But it turned out to be a really good algorithm and it had me sitting for quite some time... MD5 is at first a scary enemy, but once you get into it actually not that terrifying!

I would like to thank:

- Nacho DJ / ARTeam for being a good friend, for pushing me over the edge and for technical and moral support. He is the true king of keygenning!!
- Shub for editing
- All ARTeam members and friends for making the forum my second home.
- My wife and beautiful son for giving me the time to do what I love – reversing !

2 KEYGENNING MD5, NIEYLANA

2.1 ABSTRACT

This tutorial will cover techniques used to keygen an application that uses MD5 in its serial verification routine. The goal of this tutorial is to help bridge the gap between the surplus of beginner tutorials and the many advanced tutorials on keygenning out there on the net. Tools used in this tutorial include:

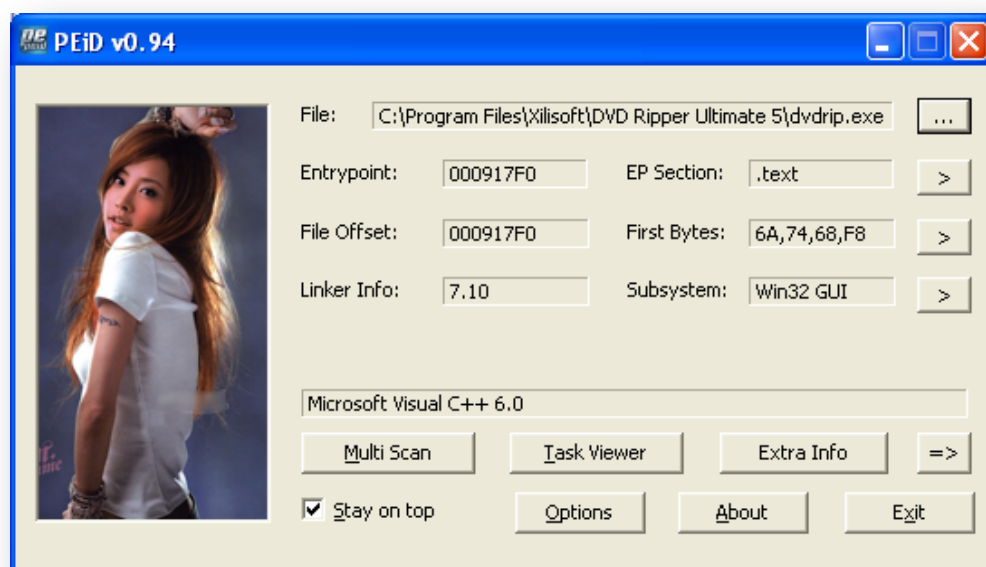
1. OllyDbg (Latest Version)
2. PEiD (with KryptoANALyzer plugin)
3. VB6 for coding Keygen.

2.2 TARGET

The target we will look at today is Xilisoft DVD Ripper Ultimate 5. Target is available for trial download at <http://www.xilisoft.com>

2.3 ANALYZING TARGET

Before we can start reversing anything, we need to first analyze the executable with PEiD to see if it protected with anything. The results are below (Hardcore Scan):

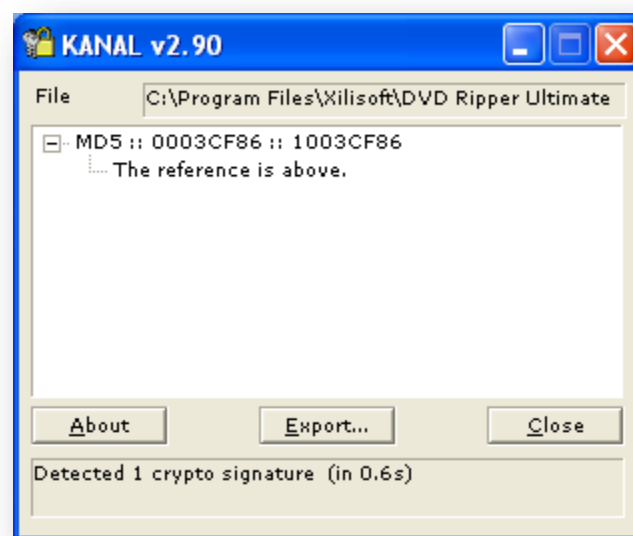


2.4 FINDING THE VERIFICATION ROUTINE

Since the target is not protected, let's open it up in OllyDbg. After it's loaded into Olly, do a search for all intermodule calls. Then type "**IsValidRegInfo**" you should see this:

00424504	CALL [C:\UIlib71U.?\IsValidRegInfo@InRegDlgEx@@SAHXZ]]	UILib71U.?\IsValidRegInfo@InRegDlgEx@@SAHXZ
004262A4	CALL [C:\UIlib71U.?\IsValidRegInfo@InRegDlgEx@@SAHXZ]]	UILib71U.?\IsValidRegInfo@InRegDlgEx@@SAHXZ
00426F71	CALL ESI	UILib71U.?\IsValidRegInfo@InRegDlgEx@@SAHXZ
00427876	CALL EBX	UILib71U.?\IsValidRegInfo@InRegDlgEx@@SAHXZ

We now know the routine is NOT in the executable but rather in the UIlib71U.dll, so we should do a scan on this DLL with PEiD to check for protectors, and crypto/hash routines. Here 's the result of that:

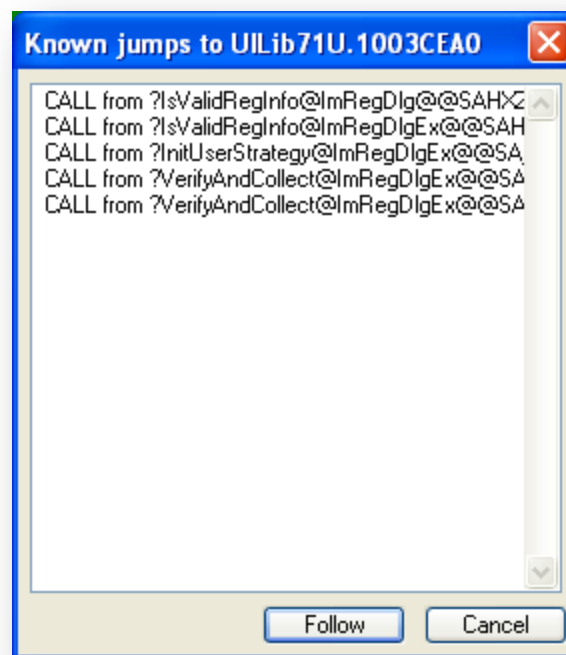


The DLL is coded in Microsoft Visual C++ 7.1, and is not protected with anything. We can also see that MD5 has been detected in the DLL at offset 0003CF86. We cannot go by 1003CF86, because when the application loads the DLL it may load it at a different address, to see where it's loaded just open up the Memory Map. On my computer it's loaded correctly at 1003CF86.

2.5 FINDING THE BEGINNING OF THE MD5 ROUTINE

The KANAL Plugin for PEiD gives us the location that if found a specific MD5 signature, it does not place us at the beginning of the routine, so we must step out a few procedures to find where the Registration routine calls it. So begin by going to the address listed by KANAL (1003CF86), and scroll up to 1003CF00, as you can see it's called from 2 different places, let's follow the first, go to 1003D843, scroll up to the beginning of this procedure. This function is called from 3 different places, again we are going to follow the first reference. Go to the address 1003CE27.

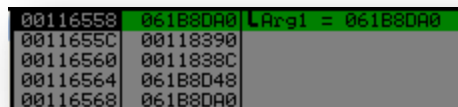
Scroll again to the top of this procedure, it's only called from 1 place, follow this one as well, and scroll to the top of the procedure. This routine is called from quite a few places as seen here:



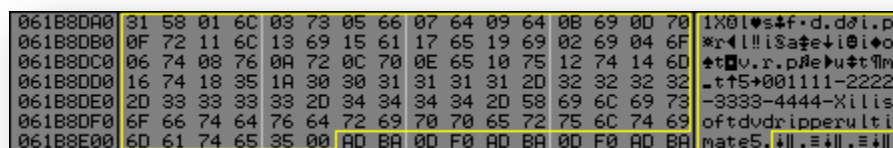
Place breakpoint on both calls from the IsValidRegInfo routine, so we can tell when the key verification routine calls the MD5 routine, and so we can see what string it's trying to hash, and lastly what it does with this hash.

2.6 FINDING THE STRING BEING HASHED

So to clarify, place BPs on these addresses: 10028504 and 1002BAD3, now make sure these are the only BPs set, and run the application and try to register with a key. I will use 1111-2222-3333-4444-5555-6666-7777-8888 After you press on the OK button OllyDbg should break on the 2nd BP that was set (1003BAD2), as we can figure, the validation routine has taken the key we gave it, and is now performing a hash on something. Here is a picture of the stack at this point:



Follow this address in the Dump The dump of the argument:



This is what the application is passing to the MD5 routine. Notice anything special about the argument being passed? This argument can be broken into 3 sections, effectively this is 3 strings that have been concatenated. The second string (middle one) is the first half of the serial number we entered followed by a hyphen. Immediately after this string, is the applications internal name „Xilisoftdvdriperultimate”.

As for the data placed before the part of our serial number, it is derived from the applications internal name, such that it begins with an ASCII „1 and then proceeds to alternate placing all odd positioned characters of the internal name and their location in the name. After they do the odd positioned letters, they then do the evens, followed finally by 2 ASCII 0s. Notice this string is also terminated by a NULL (00h) this NULL is NOT included in the MD5 hash. Let’s step this function call with F8 and see the result of this routine.

2.7 USE OF THE MD5 HASH

Now that we know what the application is hashing, our next goal is to find out how it’s using the hash. After you have stepped over the call to generate the MD5 hash, step over the next one, it will then have placed the hash into EAX, note this down. If you used my example key, your hash will be “b006c91cad46ed1261e4e767432d40ab”, ahead there is a loop, you can either trace through it, or set a BP after the loop (at address 1002BB51), Next you will need to step over the next MFC call, this is going to capitalize the result of the loop in ECX.

The result of the loop is **"B0C1-A4E1-6EE6-424A"** now compare this to the MD5 hash that was generated, it seems they have only taken 16 of the 32 bytes (every odd positioned byte) and separated them into groups of 4 with hyphens (just like your serial number). Keep stepping to see what it does next.

At address 1002BB7A you can see that both the first half of our serial and this new string, step this MFC call and then the next one (At address 1002BB9C). You can see that it put both strings together. This right here is the correct serial number; if you were to place this string in and try to register it would say you registered successfully, go ahead and try.

2.8 SUMMARY

Here I will briefly summarize everything we have learned thus far.

1. Xilisoft Applications use MD5 in the generation of valid serial numbers.
2. Xilisoft serials are 39d characters in length (8 groups of 4 with 7 hyphens)
3. The first half of the serial number is/can be completely random.
4. Xilisoft uses the last half of the serial as a type of checksum against the first half
5. The first half of the serial number is placed in the middle of 2 predefined strings before hashing, this is done most likely to minimize the possibility of brute forcing the hash.
6. The resulting hash is then capitalized.
7. They then take every odd positioned character of the hash (16d bytes) and place them together, and separate them into 4 groups of 4 with hyphens.
8. This is the checksum, or the last half of the serial number.

So with this knowledge it would be very easy to create a keygen for these types of applications, you would need a routine to generate the first half of the serial, randomly. Then a routine to place the first half of this into the middle of the 2 predefined strings. Next you'd need an MD5 hash routine, and lastly a short routine to pull every other character from the hash, and format it like a serial, then place the 16 randomly generated bytes, with the checksum and you have a serial.

2.9 GREETINGS

Special thanks goes out to ARTeam Trial Member 2kAD for allowing the pairing of our tutorials together, thanks mate!

Greetings fly out to my many friends at ARTeam, especially SSIEVIN, Shub-Niggurath, and Nacho_DJ. There are many others out there that I owe a lot to, not only for helping with ideas on Xilisoft apps, but for helping me learn RE in general. As far as teams go, ARTeam, SnD, and members of the former TeamICU.

2.10 CONCLUSION

In this tutorial we have learned that applications that use hashing are not necessarily that hard just because they do. It depends completely on the implementation of it. I have tried to design this paper so that it will help bridge the gap between Beginner tutorials, and Advanced Tutorials of keygenning. If you have any questions about my paper, or recommendations, I would ask that you contact me via the ARTeam Forums, either by posting a topic, or via PM.