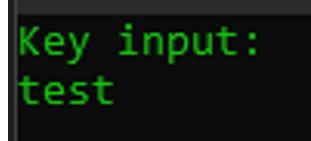


# VladMetz's CTF #5

Friday, February 18, 2022 8:22 AM

We download and run the crackme and it looks like it's asking for a key



and when the wrong key is input the program exits

We look through the string in Ghidra to find a good starting point

0042876c	Key input:	"Key input: "	ds
00428794	This is a valid key :) good job	"This is a valid key :) good job"	ds

We find an interesting string as we trace backwards from "This is a valid key :)"

004021b5 03 c1 ADD EAX, ECX	
004021b7 3d f2 9a CMP EAX, 0x9af2	
00 00	
004021bc 0f b5 03 JNZ LAB_004022c5	
01 00 00	
004021c2 ba 78 87 MOV EDX, s>You're_on_the_right_track_00428778 = "You're on the right track"	
42 00	

It looks like at least part of the comparison is a number....

004021b5 03 c1 ADD EAX, ECX	
004021b7 3d f2 9a CMP EAX, 39636	
00 00	

We run the program in ImmunityDebugger and find the point where the program requests user input and we set a couple breakpoints

00792143 . B8 6C27C008 HOU EDX, ctf_flag.007C876C	ASCII "Key input: "
00792148 . C745 FC 0000HOU DWOR PTR SS:[EBP-4], 0	
00792149 . B8 B0400000 CALL ctf_flag.007C876C	
0079214A . 50 PUSH EDX	
0079214B . E8 86070000 CALL ctf_flag.007C87E0	[Arg1 = 0077F748 ctf_flag.007C87E0
00792150 . 33C0 XOR EDX, EDX	
00792151 . 33C0 XOR ECX, ECX	
00792152 . 33C0 XOR EBX, EBX	

After some playing around this is the part of the input management that interests us

00792153 . C645 08 00 HOU BYTE PTR SS:[EBP-28], 8	
00792154 . E8 ED020000 CALL ctf_flag.007R2340	
00792155 . B8 6C27C008 HOU EDX, ctf_flag.007C876C	ASCII "Key input: "
00792156 . C745 FC 0000HOU DWOR PTR SS:[EBP-4], 0	
00792157 . B8 B0400000 CALL ctf_flag.007C876C	
00792158 . 50 PUSH EDX	
00792159 . E8 86070000 CALL ctf_flag.007C87E0	[Arg1 = 007CBF80 ASCII "LW" ctf_flag.007C87E0
0079215A . 33C0 XOR EDX, EDX	
0079215B . 33C0 XOR ECX, ECX	
0079215C . 33C0 XOR EBX, EBX	
0079215D . E8 86050000 CALL ctf_flag.007C87E0	
0079215E . 33C0 XOR EDX, EDX	
0079215F . 33C0 XOR ECX, ECX	
00792160 . 33C0 XOR EBX, EBX	
00792161 . E8 86050000 CALL ctf_flag.007C87E0	
00792162 . 33C0 XOR EDX, EDX	
00792163 . 33C0 XOR ECX, ECX	
00792164 . 33C0 XOR EBX, EBX	
00792165 . E8 86050000 CALL ctf_flag.007C87E0	
00792166 . 33C0 XOR EDX, EDX	
00792167 . 33C0 XOR ECX, ECX	
00792168 . 33C0 XOR EBX, EBX	

Inputting a long key gets some interesting results

Key input:	
thisisatestkeythatimusingtoseehowinputismanaged	

006FFFE4 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
006FFFE5 08 7C 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
006FFFE6 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
006FFFE7 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

I manipulated registers to continue to the program to try to understand what was going on

00792295 ? 0F2E05 E08277 UC0HSS XMM0, DWOR PTR DS:[7C87E0]	
00792296 . 9F LWP	
00792297 . F6C4 44 TEST AH, 44	
00792298 . 7A 19 JPEZ PTR ctf_flag.007A22C5	ccccc...

- **LAHF** instruction loads lower byte of the **EFLAGS** register into **AH** register.
- The lowest 8 bits of the flags are transferred:
  - Sign
  - Zero
  - Auxiliary Carry
  - Parity
  - Carry

- Instruction format:

```
lahf
```

So it looks like passing the first key check

```
CMP    DWORD PTR SS:[EBP-18], 0C
MOV    EBX, DWORD PTR SS:[EBP-28]
JB     ctf_flag.007A22D2
CMP    DWORD PTR SS:[EBP-14], 10
LEA    EAX, DWORD PTR SS:[EBP-28]
CMOUNB EAX, EBX
MOV    EBX, EAX
```

is a matter of setting certain values at certain points relative to ebp  
with these values

**Key input:**  
abcdefghijklmnopqrstuvwxyz0123456789

We get this

0	B8	68	CA	00	00	20	8C	00	.h <sup>±</sup> .. i.
8	08	7C	CA	00	65	66	67	68	.! <sup>±</sup> .efgh
0	69	6A	6B	6C	6D	6E	6F	00	ijklmno.
2	24	00	00	00	2F	00	00	00	\$ ./

We know from combing through this program that we need...

0063FEF0	B8	68	CA	00	00	20	8C	00	.h <sup>±</sup> .. i.
0063FEF8	08	7C	CA	00	65	66	67	68	.! <sup>±</sup> .efgh
0063FF00	69	6A	6B	6C	6D	6E	6F	00	ijklmno.
0063FF08	24	00	00	00	2F	00	00	00	\$ ./

equal to 0x0C

and...

0063FEF0	B8	68	CA	00	00	20	8C	00	.h <sup>±</sup> ..
0063FEF8	08	7C	CA	00	65	66	67	68	.! <sup>±</sup> .e
0063FF00	69	6A	6B	6C	6D	6E	6F	00	ijkl
0063FF08	24	00	00	00	2F	00	00	00	\$ ./

equal to 0x10

Next we need to leverage the arithmetic operations in order to get the value  
0x9af2

007A216A	.	837D E8 0C	CMP	DWORD PTR SS:[EBP-18], 0C
007A216E	.	8B5D D8	MOU	EBX, DWORD PTR SS:[EBP-28]
007A2171	.	0F82 5B01000	JB	ctf_flag.007A2202
007A2177	.	837D EC 10	CMP	DWORD PTR SS:[EBP-14], 10
007A217B	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A217E	.	0F43C3	CMOVNB	EAX, EBX
007A2181	.	0FBE78 06	MOVSX	EDI, BYTE PTR DS:[EAX+6]
007A2185	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A2188	.	0F43C3	CMOVNB	EAX, EBX
007A218B	.	0FEE70 07	MOVSX	ESI, BYTE PTR DS:[EAX+7]
007A218F	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A2192	.	0F43C3	CMOVNB	EAX, EBX
007A2195	.	0FBE50 09	MOVSX	EDX, BYTE PTR DS:[EAX+9]
007A2199	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A219C	.	0F43C3	CMOVNB	EAX, EBX
007A219F	.	0FBE48 0A	MOVSX	ECX, BYTE PTR DS:[EAX+A]
007A21A3	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A21A6	.	0F43C3	CMOVNB	EAX, EBX
007A21A9	.	0FAFCE	IMUL	ECX, ESI
007A21AC	.	0FBE40 0B	MOVSX	EAX, BYTE PTR DS:[EAX+B]
007A21B0	.	03C7	ADD	EAX, EDI
007A21B2	.	0FAFC2	IMUL	EAX, EDX
007A21B5	.	03C1	ADD	EAX, ECX
007A21B7	.	8D F29A0000	CMP	EAX, 9AF2

The first check

• E8 860E0000	CHLL	ctf_flag.007A2202
• 837D E8 0C	CMP	DWORD PTR SS:[EBP-18], 0C
• 8B5D D8	MOU	EBX, DWORD PTR SS:[EBP-28]
• 0F82 5B01000	JB	ctf_flag.007A2202

uses the JB instruction after the comparison is made

According to [Intel's Software Developer's Manual](#) "JB" stands for "Jump if Below"

**5.1.7 Control Transfer Instructions** The control transfer instructions provide jump, conditional jump, loop, and call and return operations to control program flow.

- JE/JZ Jump if equal/Jump if zero
- JNE/JNZ Jump if not equal/Jump if not zero
- JA/JNBE Jump if above/Jump if not below or equal
- JAE/JNBE Jump if above or equal/Jump if not below
- JB/JNAE Jump if below/Jump if not above or equal
- JBE/JNA Jump if below or equal/Jump if not above
- JG/JNL Jump if greater/Jump if not less or equal
- JGE/JNL Jump if greater or equal/Jump if not less
- JL/JNGE Jump if less/Jump if not greater or equal
- JLE/JNG Jump if less or equal/Jump if not greater
- JC Jump if carry
- JNC Jump if not carry
- JO Jump if overflow
- JNO Jump if not overflow
- JS Jump if sign (negative)
- JNS Jump if not sign (non-negative)
- JPO/JNP Jump if parity odd/Jump if not parity
- JPE/JP Jump if parity even/Jump if parity

So...if we want to continue with our instruction, we only need to make sure that the value at EBP-18 is larger than 0C, which should be pretty easy as all ASCII characters will be

The next check is going to take us into the arithmetic

007A2177	.	837D EC 10	CMP	DWORD PTR SS:[EBP-14], 10
007A217B	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A217E	.	0F43C3	CMOVNB	EAX, EBX
007A2181	.	0FBE78 06	MOVSX	EDI, BYTE PTR DS:[EAX+6]
007A2185	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A2188	.	0F43C3	CMOVNB	EAX, EBX
007A218B	.	0FBE70 07	MOVSX	ESI, BYTE PTR DS:[EAX+7]
007A218F	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A2192	.	0F43C3	CMOVNB	EAX, EBX
007A2195	.	0FBE50 09	MOVSX	EDX, BYTE PTR DS:[EAX+9]
007A2199	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A219C	.	0F43C3	CMOVNB	EAX, EBX
007A219F	.	0FBE48 0A	MOVSX	ECX, BYTE PTR DS:[EAX+A]
007A21A3	.	8D45 D8	LEA	EAX, DWORD PTR SS:[EBP-28]
007A21A6	.	0F43C3	CMOVNB	EAX, EBX
007A21A9	.	0FAFCE	IMUL	ECX, ESI
007A21AC	.	0FBE40 0B	MOVSX	EAX, BYTE PTR DS:[EAX+B]
007A21B0	.	03C7	ADD	EAX, EDI
007A21B2	.	0FAFC2	IMUL	EAX, EDX
007A21B5	.	03C1	ADD	EAX, ECX

ctf\_flag.007BE300

We start with a CMP, but instead of using it to jump, we use it to update the flags that are going to determine what arithmetic operations are conducted on our input

## The instruction CMOVNB

## Assembler:Commands:CMOVNB

**command** `cmoveb destination, source`

`cmovnb` is `CMOVcc` instruction, The `cmovnb` *conditional move if not below* check the state of **CF**.

If **CF=0** then condition is satisfied, otherwise it will be skipped.

CMOVcc instructions are mainly used after CMP

Basically, CMOVB is checking if the carry flag is set to 0, and if it is then it moves from source to destination

The key input values that matter to us for this portion are the 6th, 7th, 9th, 10th, and 11th values. We want the arithmetic to equal 39666 at the end of this

- 6 - q
- 7 - w
- 9 - e
- 10 - r
- 11 - t

```
CMP    DWORD PTR SS:[EBP-14], 10
LEA    EAX, DWORD PTR SS:[EBP-28]
COUNB  EBX
MOVSX  EDI, BYTE PTR DS:[EAX+6]
LEA    EAX, DWORD PTR SS:[EBP-28]
COUNB  EBX
MOVSX  ESI, BYTE PTR DS:[EAX+7]
LEA    EAX, DWORD PTR SS:[EBP-28]
COUNB  EBX
MOVSX  EDX, BYTE PTR DS:[EAX+9]
LEA    EAX, DWORD PTR SS:[EBP-28]
COUNB  EBX
MOVSX  ECX, BYTE PTR DS:[EAX+A]
LEA    EAX, DWORD PTR SS:[EBP-28]
COUNB  EBX
IMUL   ECX, ESI
MOVSX  EAX, BYTE PTR DS:[EAX+B]
ADD    EAX, EDI
IMUL   EAX, EDX
ADD    EAX, ECX
CMP    EAX, 9AF2
```

The value we want is 39666 in decimal

The equations looks something like

$$ECX = r^*w$$

$$EAX = t+q$$

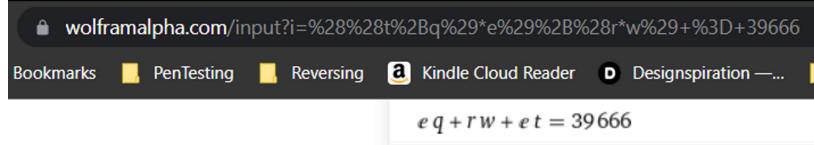
$$EAX = (t+q)^*e$$

Final

$$EAX = ((t+q)^*e) + (r^*w)$$

where EAX MUST = 39666

Using wolframalpha we get a more clear version



$$eq + rw + et = 39666$$

The minimum value that I can enter with a keyboard into the program is 0x20 (space) and that's 32 in decimal and the largest I care about entering is the tilde (7e hex, 126 decimal)

0th	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th
0	0	0	0	0	0	q	w	0	e	r	t

Doing a little bit of math, we can reduce the keyspace a little bit  
Assuming max values for the other variables, it's not possible to generate a  
number > 39875 until q is equal to 70, so we can write some code to leverage this  
and find us some keys (after some trial and error I found that keys don't start  
popping up into q=85

\*\*note, mathematically it doesn't matter which of these variable is 85, I chose q  
arbitrarily\*\*

Using the below code we brute force value that are going to pass these checks

```
#!/usr/bin/env python3

__author__ = "triboulet"

q = []
w = []
e = []
r = []
t = []

def generate_q(a):
    for i in range(85,126):
        a.append(i)

def generate(a):
    for i in range(33,126):
        a.append(i)
    print("generating lists")

generate_q(q)
generate(w)
generate(e)
generate(r)
generate(t)

print("generated...")
print("q=", q)
print("w=", w)
print("e=", e)
print("r=", r)
print("t=", t)

print("beginning operation: Big Fucking Hammer")

old_i = 0

for i in range(0,len(q)-1):
    for j in range(0,len(w)-1):
        for k in range(0,len(e)-1):
            for l in range(0,len(r)-1):
                for m in range(0,len(t)-1):
                    final = e[k]*q[i] + r[l]*w[j] + e[k]*t[m]
                    #if(i > old_i): ## arbitrary if statement, validates program continuing to
run
                    #print(final)
                    #old_i += 1
                    if final == 39666:
                        print("VALID CASE:\n")
                        print("q=", q[i])
                        print("w=", w[j])
                        print("e=", e[k])
                        print("r=", r[l])
                        print("t=", t[m, end="\n\n\n"])
```

One of the test cases generated is

```
88 124 118 121 121
```

## Output

```
x|vy
```

0th	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th
0	0	0	0	0	0	X		0	v	y	y

```
000000X|0vy
```

and it looks like we get the value that we want in eax

Registers (MMX)	
EAX	00009AF2
ECX	00003A9C
EDX	00000076
EBX	00B97C08 ASCII "000000X 0vy
ESP	0073FA48
EBP	0073FA80
ESI	0000007C
EDI	00000058

and we get the message that we want

C:\Users\ssali\Downloads\6207e29133c5d46c8bcbfdac\ctf_flagmeNEW.exe
Key input:
000000X 0vy000000X 0vy000000X 0vy
You're on the right track

Now onto the next series of checks

## The Second Check

After we pass the arithmetic checks, our key gets passed to a function that prints and returns our input with some manipulations, this then gets passed onto another function

004521C2	. BA 78874700	HOU EDX, ctf_flag.00478778	ASCII "You're on the right track"
004521C7	: EB 44040000	CALL ctf_flag.00452610	
004521CD	: SE 0E070000	PUSH EBX	Reg1 = 0047BF80 ASCII "Lnf"

fun.00452610 => printAndChange

Stepping through this function, we see that part of our key is going to end up on the stack...for the sake of clarity I'm going to remake our input

1	:	63	PUSH EBX	
0	:	68	PUSH ESI	
0	:	67	PUSH EDI	

00D9F060	00000058	X...
00D9F064	0000007C	I...
00D9F068	30303030	0000

0th	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th
0	1	2	3	4	5	X		8	v	y	y

```
012345X|8vy
```

Now we know that at this point our input is on the stack as follows

```
00452624 : 53 PUSH EBV  
00452625 : 56 PUSH ESI  
00452626 : 57 PUSH EDI  
00452627 . R1 14804700 NOV ERX, DDWORD PTR DS:[478014]
```

008FF7A0	00000058	X...
008FF7A4	0000007C	I...
008FF7A8	33323130	0123

Next we see our first operation

```
33C5 XOR EAX, EBP
```

At this point our registers look like this

```
Registers (FPU)  
ERX E11578E1  
ECX 00003A9C  
EDX 00478778 ASCII "You're on the right track"  
EBX 33323130  
ESP 008FF7A0  
EBP 008FF7DC  
ESI 0000007C  
EDI 00000058
```

Since EBP is pointing at X we know that at this point, the program is going to XOR some static data with our 6th value

It looks like this XOR happens, then the program starts prepping to print "You're on the right track"

```
008FF79C E19A8C3D =ÜB  
008FF7A0 00000058 X...  
008FF7A4 0000007C I...  
008FF7A8 33323130 0123  
008FF7AC 0047C000 "L.G. ctf_flag.0047C000  
008FF7B0 009B76A8 .V.  
008FF7B4 008FF7D8 T.A.  
008FF7B8 00478778 x.G. ASCII "You're on the right track"  
008FF7BC 0047C000 "L.G. ctf_flag.0047C000  
008FF7C0 7FFFFFF3 £...  
008FF7C4 01000001 ....  
008FF7C8 009B7679 y.V..  
008FF7CC 008FF79C £.A. ASCII "=ÜBX"
```

```
Registers (FPU)  
EAX 00478778 ASCII "You're on the right track"  
ECX 00003A9C  
EDX 00478778 ASCII "You're on the right track"  
EBX 33323130  
ESP 008FF79C ASCII "=ÜBX"  
EBP 008FF7DC  
ESI 0000007C  
EDI 00000058  
EIP 00452640 ctf_flag.00452640
```

If we skip through the whole function until we return to main, then we end up with an arbitrary EAX getting passed into the next function

```
008FF7E0 0047BF80 .7.G. LArg1 = 0047BF80 ASCII "Ltf"  
008FF7E4 E19A83ED 3 ÜB
```

Skipping forward a few steps...it doesn't look like any of these checks matter.

Ultimately the next comparison happens here

```
00402200 7a 13 JP LAB_004022c5  
004022b2 ba 94 87 MOV EDX,s_This_is_a_valid_key_:_good_job_004...= "This is a valid key :)" go...  
42 00  
004022b7 e8 54 03 CALL printAndChange undefined4 * printAndChange(...  
00 00  
004022bc 50 PUSH EAX  
004022bd e8 1e 06 CALL FUN_004028e0 int * FUN_004028e0(int * par...  
00 00  
004022c2 83 c4 04 ADD ESP,0x4  
  
LAB_004022c5  
004022c5 48 83 ec 20 sub esp, 20  
XREF[2]: 004021bc(j), 004022b0(j)
```

And it looks like ImmunityDebugger tells us that this is a JPE instruction

```
R1 19 [JPE] EDX,ctf_flag.00478794 ASCII "This is a valid key :)" good job"  
R8 94874700 NOV EDX,ctf_flag.00452610 ASCII "You're on the right track"  
E8 54030000 CALL ctf_flag.00452610 [Arg1 = 008FF7F4 ASCII "012345X:Üvy"]  
E8 1E050000 PUSH EAX,ctf_flag.004526E0 [ctf_flag.004526E0]  
88C4 04 ADD ESP,4 ASCII "pause"  
68 B4874700 PUSH EDX,ctf_flag.004787B4 ASCII "pause"
```

```
JP JPE Jump if parity even  
Jump if parity even PF = 1
```

It looks like if the parity flag is set we jump over the good message, and if it is not, then we get the good message, we see this by changing the parity flag manually

In x86 processors, the parity flag reflects the parity only of the *least significant byte* of the result, and is set if the number of set bits of ones is even (put another way, the parity bit is set if the sum of the bits is even). According to 80386 Intel manual, the parity flag is changed in the x86 processor family by the following instructions:

- All arithmetic instructions;

In conditional jumps, parity flag is used, where e.g. the JP instruction jumps to the given target when the parity flag is set and the JNP instruction jumps if it is not set. The flag may be also read directly with instructions such as PUSHF, which pushes the flags register on the stack.

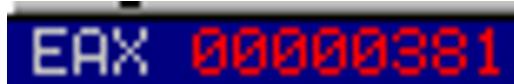
One common reason to test the parity flag is to check an unrelated FPU flag. The FPU has four condition flags (C0 to C3), but they can not be tested directly, and must instead be first copied to the flags register. When this happens, C0 is placed in the carry flag, C2 in the parity flag and C3 in the zero flag.<sup>[1]</sup> The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.

So it looks like the TEST before this comparison is being used to update the Parity

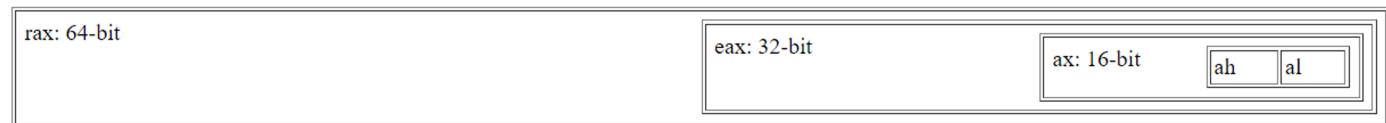
Flag

In the x86 assembly language, the TEST instruction performs a *bitwise AND* on two operands. The flags SF, ZF, PF are modified while the result of the AND is discarded. The OF and CF flags are set to 0, while AF flag is undefined. There are 9 different opcodes for the TEST instruction depending on the type and size of the operands. It can compare 8-bit, 16-bit, 32-bit or 64-bit values. It can also compare registers, immediate values and register indirect values.<sup>[1]</sup>

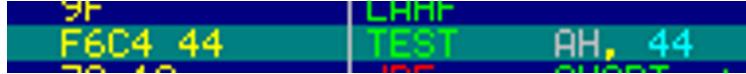
At the moment that our test happens, this is the EAX register



Remembering this



We know this is going to set the parity flag because AH = 03



So we need to figure out how the program is setting the value of AH so that we can change our input to match

Remember our friend LAHF?

- LAHF instruction loads lower byte of the EFLAGS register into AH register.
- The lowest 8 bits of the flags are transferred:
  - Sign
  - Zero
  - Auxiliary Carry
  - Parity
  - Carry
- Instruction format:

lahf

Before LAHF

# EFL 00000203 (NO,B,NE,BE,NS,PO,GE,G)

After LAHF

CPU Registers				Registers (MM)
00452240 : 9F	LAHF	AH, 44	TEST	ECX 00000031 ECX 00000035 EDX 000002A4
00452280 : F6C4 44	JPE	AH, 44	SHORT ctf_flag.004522C5	
004522B0 : 7A 13	MOV	ctf_flag.004522C4	ASCII "This is a valid key :) good job"	
004522D0 : 80 9A 74 700	MOV	ctf_flag.004522C4		

So it looks like we need to decrement the G flag

Looking at the stack at this point

00EFF7BC	FAD05579	yUW.
00EFF7C0	01188388	...
00EFF7C4	01186750	Pg..
00EFF7C8	0006A000	..rr.
00EFF7CC	33323130	0123
00EFF7D0	7C583534	45X!
00EFF7D4	79797638	8vyy
00EFF7D8	00EFF700	..n.
00EFF7DC	0000000C	....
00EFF7E0	0000000F	...::
00EFF7E4	FAD05579	yUW.
00EFF7E8	00EFF82C	^n.
00EFF7EC	0046D7FD	^ .F.
00EFF7F0	00000000	....

I'm getting a little confused tracking down how some of the stuff is happening because we have two ys in our key...lets generate another one using our cracker.py script

```
VALID CASE:
q= 89
w= 124
e= 118
r= 121
t= 120
```

```
q=89
w=124
e=118
r=121
t=120
```

0th	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th
0	1	2	3	4	5	q	w	8	e	r	t

0th	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th
0	1	2	3	4	5	Y		8	v	y	x

012345Y|8vyx

What's interesting about generating this new key...is that our EAX value at the LAHF point is the same...

CPU Registers				Registers (MM)
00452240 : 9F	TEST	AH, 44	JPE	ERX 00000731 ECX 00000031 EDX 000002A4
00452280 : F6C4 44	JPE	AH, 44	SHORT ctf_flag.004522C5	
004522B0 : 7A 13	MOV	ctf_flag.004522C4	ASCII "This is a valid key :) good job"	
004522D0 : 80 9A 74 700	MOV	ctf_flag.004522C4		

Since we know what values need to remain constant in order to get to this point, lets play around with some of the other values in order to try to drive a change in EAX

00452220	D1	DB	D1	
00452230	83	DB	83	
0045223F	7C	DB	7C	CHAR 'Y'
00452240	10	DB	10	
00452241	67	DB	67	
00452242	43	DB	43	CHAR 'C'
00452243	C8	DB	C8	CHAR 'F'
00452244	55	DB	55	
00452245	9F	DB	9F	CHAR 'n'
00452246	0D	DB	0D	
00452247	10	DB	10	
00452248	D2	DB	D2	
00452249	F3	DB	F3	
0045224D	67	DB	67	
0045224E	E6	DB	E6	
0045224F	DB	DB	DB	
00452250	9E	DB	9E	
00452251	BE	DB	BE	
00452252	05	DB	05	CHAR 'I'
00452253	80	DB	80	
00452254	03	DB	03	
00452255	9F	DB	9F	
00452256	8F	DB	8F	
00452257	07	DB	07	
00452258	66	DB	66	CHAR 'f'
00452259	AE	DB	AE	CHAR 'n'
0045225A	C9	DB	C9	
0045225B	F9	DB	F9	
0045225D	9F	DB	9F	
0045225E	E6	DB	E6	
0045225F	03	DB	03	
00452260	28	DB	28	CHAR 'z'
00452261	C8	DB	C8	CHAR 'F'
00452262	54	DB	54	CHAR 'n'
00452263	9F	DB	9F	
00452265	D0	DB	D0	
00452266	F3	DB	F3	
00452267	9F	DB	9F	
00452268	E6	DB	E6	
00452269	D2	DB	D2	
0045226A	9F	DB	9F	CHAR 'v'
0045226D	CD	DB	CD	
0045226E	F2	DB	F2	
00452270	59	DB	59	CHAR 'V'
00452271	00	DB	00	
00452272	9F	DB	9F	
00452273	28	DB	28	CHAR 'Y'

\*\*Note: I had noticed this before but it hadn't mattered too much, but Immunity is garbling some of the instructions in this part of the program...I switched to x32dbg from this point\*\*

So after inputting our normal key to the program, following the printAndChange function our key gets transformed into

EAX 0047BF80 "LäF"

Key1:  
012349Y|8vyx

Key2:  
012345Yx8zr|

Key3:  
092349Yx8zr|

Key 4:  
012345YxAzr|

So after multiple attempts and modifications to the key, it looks like the value of EAX at this point is solid

	Log	Notes	Breakpoints	Memory Map	Call Stack	SEM	Script	Symbols	Source	References	Threads
● 004521B7	3D F29A0000			Cmp eax,9AF2				eax:"LäF"			
● 004521BC	0F85 03010000			je ctf_flagmenew.4522C5				edx:"?;E", 478778:"You're on t			
● 004521C2	B8 78874700			mov edx,ctf_flagmenew.478778				eax:"LäF"			
● 004521C7	E8 44040000			call ctf_flagmenew.452610							
● 004521CC	50			push eax							
● 004521CD	E8 0E070000			call ctf_flagmenew.4528E0							
● 004521D2	83C4 04			add esp,4							
● 004521D5	8D45 D8			lea eax,dword ptr ss:[ebp-28]							
● 004521D8	837D EC 10			cmp dword ptr ss:[ebp-14],10							
● 004521DC	8D4D D8			lea ecx,dword ptr ss:[ebp-28]							
● 004521DF	0F43C3			cmovae eax,ebx							
● 004521E2	0FBE30			movsx esi,byte ptr ds:[eax]				eax:"LäF"			

Stepping through the code we run into some interesting stuff

Unwind edx,ecx  
movd xmm0,esi

MM registers are the registers used by the MMX instruction set, one of the first attempts to add (integer-only) SIMD to x86. They are 64 bit wide and they are actually aliases for the mantissa parts of the x87 registers (but they are not affected by the FPU top of the stack position); this was done to keep compatibility with existing operating systems (which already saved the FPU stack on context switch), but made using MMX together with floating point a non trivial job.

Nowadays they are just a historical oddity. I don't think anybody actually uses MMX anymore, as it has been completely superseded by the various SSE extensions. *Edit:* as Peter Cordes points out in the comments, there is still quite some MMX code around.

XMM registers, instead, are a completely separate registers set, introduced with SSE and still widely used to this day. They are 128 bit wide, with instructions that can treat them as arrays of 64, 32 (integer and floating point), 16 or 8 bit (integer only) values. You have 8 of them in 32 bit mode, 16 in 64 bit. Virtually all floating point math is done in SSE (and thus XMM registers) in 64 bit mode, so, unlike MMX registers, they are still quite relevant.

Nowadays you may also meet the YMM and ZMM registers; they were introduced respectively with the AVX (2011) and AVX-512 (2015) instruction sets, and they expand the XMM registers, not unlike the `e` and `r` extensions to the general-purpose registers (`rax` extended `eax` which extended `ax` which can be accessed as `ah : al`).

In an AVX-capable processor, each register in the XMM register file is expanded to 256 bits. The whole 256 bit register is referred to as YMM $x$  ( $x$  from 0 to 15) and can be used by the new AVX instructions, the lower half is XMM $x$ , and can be still used by older SSE instructions.

Similarly, AVX-512 expands the registers above to 512 bit; the whole register is ZMMx (usable with the AVX-512 instructions), the lower 256 bit is YMMx (also usable with AVX instructions), the lower 128 bits are still XMMx (usable also with SSE). Also, the register count is increased to 32, so these registers are both bigger and twice in number.

Share Improve this answer Follow edited Jan 8, 2020 at 5:39 answered Jun 1, 2017 at 6:07

 user3096803 Matteo Italia 118K 17 192 288

A little bit further down we have this monstrosity

```
cvtdq2pd xmm0,xmm0
```

## Description

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double-precision floating-point values in the destination operand (the first operand).

**EVEX encoded versions:** The source operand can be a YMM/XMM/XMM (low 64-bit) register, a 256/128-64-bit memory location or a 256/128-64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX extended rounding is ignored.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEY<sub>1111b</sub> and EVEY<sub>1111b</sub> are reserved and must be 1111b, otherwise instructions will #UD.

VERA\_NNNN and EVEANNNN are reserved and must be 11110, otherwise instructions will =CD.

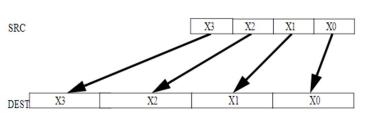


Figure 3-11. CVTDO2PD (VEX 256 encoded version)

Straight forward enough, so we should expect the 0000...30 in `xmm0` to become more significant bitwise

Original

## Results:

```
XMM0 00000000000000000000000404800000000000000  
XMM1 000000000000000000000000000000000000000000  
XMM2 000000000000000000000000000000000000000000  
XMM3 000000000000000000000000000000000000000000  
XMM4 000000000000000000000000000000000000000000  
XMM5 000000000000000000000000000000000000000000  
XMM6 000000000000000000000000000000000000000000
```

XMM7 00000000000000000000000000000000

YMM0 00000000000000000000000000000000  
Interesting...next we move the 2nd value  
XMM5 00000000000000000000000000000032

And then we have this instruction again

```
cvtdq2pd xmm5,xmm5  
movsd eax,ds:[4787D8]
```

And this is the result

```
XMM5 00000000000000004049000000000000  
XMM6 00000000000000000000000000000000  
XMM7 00000000000000000000000000000000  
  
YMM0 000000000000000000000000000000004048000000000000  
YMM1 000000000000000000000000000000000000000000000000000  
YMM2 000000000000000000000000000000000000000000000000000  
YMM3 000000000000000000000000000000000000000000000000000  
YMM4 000000000000000000000000000000000000000000000000000  
YMM5 00000000000000000000000000000000000000000000000000040490000000000000
```

Now things start to get very interesting

```
mulsd xmm5,qword ptr ds:[4787D8]
```

```
XMM5 00000000000000004039000000000000  
XMM6 00000000000000000000000000000000  
XMM7 00000000000000000000000000000000  
  
YMM0 000000000000000000000000000000004048000000000000  
YMM1 000000000000000000000000000000000000000000000000000  
YMM2 000000000000000000000000000000000000000000000000000  
YMM3 000000000000000000000000000000000000000000000000000  
YMM4 000000000000000000000000000000000000000000000000000  
YMM5 00000000000000000000000000000000000000000000000000040390000000000000
```

Then we see this

```
XMM0 00000000000000004048000000000000  
XMM1 00000000000000000000000000000000  
XMM2 00000000000000000000000000000000  
XMM3 00000000000000000000000000000000  
XMM4 00000000000000004039000000000000  
XMM5 00000000000000000000000000000000  
XMM6 00000000000000000000000000000000  
XMM7 00000000000000000000000000000000  
  
YMM0 0000000000000000000000000000000040480000000000000  
YMM1 000000000000000000000000000000000000000000000000000  
YMM2 000000000000000000000000000000000000000000000000000  
YMM3 000000000000000000000000000000000000000000000000000  
YMM4 000000000000000000000000000000000000000000000000000  
YMM5 00000000000000000000000000000000000000000000000000040390000000000000  
YMM6 000000000000000000000000000000000000000000000000000  
YMM7 000000000000000000000000000000000000000000000000000
```

Then

```
XMM0 00000000000000004048000000000000  
XMM1 00000000000000000000000000000000  
XMM2 00000000000000000000000000000000  
XMM3 0000000000000000000000000000001A  
XMM4 00000000000000004039000000000000  
XMM5 00000000000000000000000000000000  
XMM6 00000000000000000000000000000000  
XMM7 00000000000000000000000000000000  
  
YMM0 0000000000000000000000000000000040480000000000000  
YMM1 000000000000000000000000000000000000000000000000000  
YMM2 000000000000000000000000000000000000000000000000000  
YMM3 0000000000000000000000000000000000000000000000000001A  
YMM4 000000000000000000000000000000000000000000000000000  
YMM5 00000000000000000000000000000000000000000000000000040390000000000000  
YMM6 000000000000000000000000000000000000000000000000000  
YMM7 000000000000000000000000000000000000000000000000000
```

Then

```
XMM0 00000000000000004048000000000000  
XMM1 00000000000000000000000000000000  
XMM2 00000000000000000000000000000000  
XMM3 00000000000000000000000000000000  
XMM4 00000000000000004039000000000000  
XMM5 00000000000000000000000000000000  
XMM6 00000000000000000000000000000000  
XMM7 00000000000000000000000000000000  
  
YMM0 0000000000000000000000000000000040480000000000000  
YMM1 000000000000000000000000000000000000000000000000000  
YMM2 000000000000000000000000000000000000000000000000000  
YMM3 000000000000000000000000000000000000000000000000000403A00000000000000  
YMM4 000000000000000000000000000000000000000000000000000  
YMM5 00000000000000000000000000000000000000000000000000040390000000000000  
YMM6 000000000000000000000000000000000000000000000000000  
YMM7 000000000000000000000000000000000000000000000000000
```

This is the first time that numbers interact like this

	00452224	66:0F6EE0	movd xmm4, eax		
	00452228	8D45 D8	lea eax,dword ptr ss:[ebp-28]		
	0045222B	0F43C3	cmovea eax,ebx		
	0045222E	F3:0FE6E4	cvtdq2pd xmm4,xmm4		
	00452232	0FBFE40 04	movsx eax,byte ptr ds:[eax+4]		
	00452236	99	cdq		
	00452237	2BC2	sub eax,edx		
	00452239	8BD0	mov edx,eax		
	0045223B	D1FA	sar edx,1		
	0045223D	837D EC 10	cmp dword ptr ss:[ebp-14],10		
	00452241	0F43CB	cmovea ecx,ebx		
	00452244	66:0F6EDA	movd xmm3,edx		
	00452248	0FADF2	imul edx,edx		
	0045224B	F3:0FE6DB	cvtdq2pd xmm3,xmm3		
	0045224F	0FBFE49 05	movsx ecx,byte ptr ds:[ecx+5]		
	00452253	8BC1	mov eax,ecx		
EIP →	00452255	0FAFC7	imul eax,edi		
	00452258	66:0FEC9	movd xmm1,ecx		
	0045225C	F3:0FE6C9	cvtdq2pd xmm1,xmm1		

## Result

EAX	00000A25	ל.'ב'
EBX	33333130	

Final load out of the registers before any arithmetic operations

012345Yx8zr |

ucomiss xmm0,dword ptr ds:[4787E0] 004787E0:"-,3I"

That ucomiss instruction does the following

## Description

Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNAN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The FFI AGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX, EVEX, and FVEX prefixes are ignored and must be 1111h; otherwise instructions will #UD.

Note: VEX:XXXX and EVEX:XXXX are reserved and must be 11110, otherwise instructions will #UD.

So we only care about the effect this instruction has on the flags

So we only care  
Before we arrive

Before ucomiss  
EFLAGS 00000304  
ZF 0 PF 1 AF 0  
OF 0 SF 0 DF 0  
CE 0 TF 1 TE 1

After ucomiss

EFLAGS 00000203  
ZF 0 PF 0 AF 0  
OF 0 SF 0 DF 0  
CF 1 TF 0 IF 1

Before LAHE

```
before LAH  
EAX 00000781 L'`  
EBX 33323130  
ECX 00000035 '5'  
EDX 00000244 L'`  
EBP 007EF9A0 &"öù~"  
ESP 007EF968  
ESI 00000030 '0'  
EDI 00000031 '1'  
  
EIP 004522AC ctf_flagmenew.004522AC  
  
EFLAGS 00000203  
ZF 0 PF 0 AF 0  
OF 0 SF 0 DF 0  
CF 1 TF 0 IF 1
```

After LAHF

```
Attacker: LARII  
  
EAX 000000381  
EBX 33323130  
ECX 00000035 '5'  
EDX 00000244 L'君  
EBP 007EF9AO "&\"~"  
ESP 007EF968  
ESI 00000030 '0'  
EDI 00000031 '1'  
  
EIP 004522AD ctf_flagmenew.004522AD  
  
EFLAGS 00000203
```

After going through this loop a couple times, it seems to me we have a couple key pieces of arithmetic...

This sets edx

## Before

```
| sar edx,1
```

EAX	00000034	'4'
EBX	33323130	
ECX	008BF9C4	"012345Yx8zr  "
EDX	00000034	'4'
EBP	00BF9E9C	&"DÚž"
ESP	00BF9F94	
ESI	00000030	'0'
EDI	00000031	'1'
EIP	0045223B	ctf_flagmenew.0045223B

After

EAX	00000034	'4'
EBX	33323130	
ECX	00BF9C4	"012345Yx8zr  "
EDX	<b>0000001A</b>	
<u>EBP</u>	00BF9E9C	&"Dúč"
ESP	00BF9E94	
ESI	00000030	'0'
EDI	00000031	'1'

Next we multiply  $edx$  by itself

imul edx, edx

## Result

EDX 000002A4 L'ñ'

Next we take the 5th term and multiply it with that result and store that in EAX

imul eax,edi

```
EAX 00000A25 L'曰'  
EBX 33323130  
ECX 00000035 '5'  
EDX 000002A4 L'あ'  
EBP 00BF79EC "&DUZ"  
ESP 00BF7984  
ESI 00000030 '0'  
EDI 00000031 '1'  
  
EIP 00452258 ctf_flagmenew.00452258
```

Then we subtract the EAX-EDX and store it in EAX

EAX	00000A25	L'�'
EBX	33323130	'5'
ECX	00000035	'5'
EDX	000002A4	L'�'
EBP	00BFF9EC	"�D�"
ESP	00BFF984	
ESI	00000030	'0'
EDI	00000031	'1'
EIP	00452260	ctf_flagmenew.00452260

EAX	00000781	L'ゝ'
EBX	33323130	
ECX	00000035	'5'
EDX	000002A4	L'ゞ'
EBP	00BF9ECC	"&DÙゞ"
ESP	00BF9B4	
ESI	00000030	'0'
EDI	00000031	'1'

But at the end of the day, the only thing we need to understand out of all of this, is that we need to change the flags that get set during the `xmmX` multiplication because that's what ends up getting loaded into AH and therefore the parity comparison that we need to bypass

Using the keys I've generated gets me these flags

EFLAGS	00000203				
ZF	0	PF	0	AF	0
OF	0	SF	0	DF	0
CF	1	TF	0	IF	1

After some trial and error, these are the flags that I want:

EFLAGS	00000204				
ZF	0	PF	1	AF	0
OF	0	SF	0	DF	0
CF	0	TF	0	IF	1

0th	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th
0	1	2	3	4	5	Y	I	8	v	y	x

~~~~~X|8vyy

Key to our success is the following instructions:

|          |                 |                                                                                                   |                 |
|----------|-----------------|---------------------------------------------------------------------------------------------------|-----------------|
| 0045229D | F2:0F58D3       | addsd xmm2,xmm3<br>cvtpd2ps xmm0,xmm2<br>ucomiss xmm0,dword ptr ds:[4787E0]<br>lahf<br>test ah,44 | 004787E0:"~,3I" |
| 004522A1 | 66:0F5AC2       |                                                                                                   |                 |
| 004522A5 | 0F2E05 E0874700 |                                                                                                   |                 |
| 004522AC | 9F              |                                                                                                   |                 |
| 004522AD | F6C4 44         |                                                                                                   |                 |

And this value

Address Hex  
004787E0 AC 2C 33 49

004787E0 AC 2C 33 49 00 00 00 00 04 E2 07 62 00 00 00 00 ~,3I....â.b....

We need to generate input that results in  
0x49332cac (1228090540 in decimal and 733899 float)

It's really easy to get caught up in the cvtdq2pd function, but it's just a type conversion so we can pretty much ignore it in our python code

Below is the value of the first term in our key "~~" after the conversion, we can see that the value "126" is the same as the decimal value but it's stored as a double

Edit YMM register

|                                                                                                          |                                  |            |       |       |       |       |       |    |
|----------------------------------------------------------------------------------------------------------|----------------------------------|------------|-------|-------|-------|-------|-------|----|
| High:                                                                                                    | 00000000000000000000000000000000 |            |       |       |       |       |       |    |
| Byte:                                                                                                    | 10-11                            | 12-13      | 14-15 | 16-17 | 18-19 | 1A-1B | 1C-1D | 1E |
| Word:                                                                                                    | 0                                | 0          | 0     | 0     | 0     | 0     | 0     | 0  |
| Dword:                                                                                                   | 0                                | 0          | 0     | 0     | 0     | 0     | 0     | 0  |
| Float:                                                                                                   | 0                                | 0          | 0     | 0     | 0     | 0     | 0     | 0  |
| Double:                                                                                                  | 0                                | 0          | 0     | 0     | 0     | 0     | 0     | 0  |
| Qword:                                                                                                   | 0                                | 0          | 0     | 0     | 0     | 0     | 0     | 0  |
| Low:                                                                                                     | 0000000000000000405F800000000000 |            |       |       |       |       |       |    |
| Byte:                                                                                                    | 0-1                              | 2-3        | 4-5   | 6-7   | 8-9   | A-B   | C-D   | E  |
| Word:                                                                                                    | 0                                | -32768     | 16479 | 0     | 0     | 0     | 0     | 0  |
| Dword:                                                                                                   | 0                                | 1080000512 | 0     | 0     | 0     | 0     | 0     | 0  |
| Float:                                                                                                   | 0                                | 3.49219    | 0     | 0     | 0     | 0     | 0     | 0  |
| Double:                                                                                                  | 126                              | 0          | 0     | 0     | 0     | 0     | 0     | 0  |
| Qword:                                                                                                   | 4638566878703255552              | 0          | 0     | 0     | 0     | 0     | 0     | 0  |
| <input type="radio"/> Hexadecimal <input checked="" type="radio"/> Signed <input type="radio"/> Unsigned | OK                               | Cancel     |       |       |       |       |       |    |

There IS a bunch of arithmetic we need to keep track of so let's break that down

Given the key:

012345X|8vyv

|         | 0th | 1st | 2nd | 3rd | 4th | 5th | 6th | 7th | 8th | 9th | 10th | 11th |
|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
|         | 0   | 1   | 2   | 3   | 4   | 5   | X   |     | 8   | v   | y    | y    |
| decimal | 48  | 49  | 50  | 51  | 52  | 53  |     |     |     |     |      |      |

At the start---

xmm0 - 48 (0th term)  
xmm1 - 53 (5th term)  
xmm2 - 1921 ((multiply 5th and 1st term)-(4th term shift right squared))  
xmm3 - 26 (fourth term, arithmetic shift right 1)  
xmm4 - 25 (third term, arithmetic shift right 1)  
xmm5 - 25 (second term /2)

```

xmm1 = xmm1 * xmm5
xmm2 = xmm2 * xmm0
xmm0 = xmm3
xmm0 = xmm0 * xmm4
xmm3 = xmm3 * xmm5
xmm1 = xm1 - xmm0
xmm0 = 1st term
xmm1 = xmm1 * xmm5
xmm0 = xmm0 * xmm4
xmm2 = xmm2 - xmm1
xmm3 = xmm3 - xmm0
xmm3 = xmm3 * xmm4
xmm2 = xmm2 + xmm3
xmm0 = xmm2 = 1228090540

```

We use the following script to try to crack some of the key values

#!/usr/bin/env python3

```

__author__ = "triboulet"

max_num = 0
flag = 0
for zero in range(126,33,-1):
    for one in range(126,33,-1):
        for two in range(126,33,-1):
            for three in range(126,33,-1):

```

```

for four in range(126,33,-1):
    for five in range(126,33,-1):
        xmm0 = zero
        xmm1 = five
        xmm2 = (five * one)-((four>>1)**2)
        xmm3 = four >> 1
        xmm4 = three >> 1
        xmm5 = two / 2
        xmm1 = xmm1 * xmm5
        xmm2 = xmm2 * xmm0
        xmm0 = xmm3
        xmm0 = xmm0 * xmm4
        xmm3 = xmm3 * xmm5
        xmm1 = xmm1 - xmm0
        xmm0 = one
        xmm1 = xmm1 * xmm5
        xmm0 = xmm0 * xmm4
        xmm2 = xmm2 - xmm1
        xmm3 = xmm3 - xmm0

        xmm3 = xmm3 * xmm4
        xmm2 = xmm2 + xmm3
        xmm0 = xmm2
        if(xmm0 > max_num and flag == 0):
            max_num = xmm0
            print()
            print("max num: ", max_num)
            print("zero: ", zero)
            print("one: ", one)
            print("two: ", two)
            print("three: ", three)
            print("four: ", four)
            print("five: ", five)
            print()

if (xmm2 == 733898.75): ## this is the weirdness caused from the
differences in type management between python and c++

```

```

if(flag == 0):
    print()
    print("*POSSIBLE* Valid cases found:")
    flag = 1
key = ""
key += chr(zero)
key += chr(one)
key += chr(two)
key += chr(three)
key += chr(four)
key += chr(five)
key += 'X|8vy'
print(key)

```

```

*POSSIBLE* Valid cases found:
~~goGQX|8vy
~~goFQX|8vy
~~gnGQX|8vy
~~gnFQX|8vy
~~gGoQX|8vy
~~gFnQX|8vy
~~gQnQX|8vy
~~gFoQX|8vy
~~gFnQX|8vy

```

The keys I tested look like:

~~goGQX|8vy

Things that can be improved:

This code would be more seamless in C++

The serial key does not use the 8th digit for anything that matters as far as I can tell