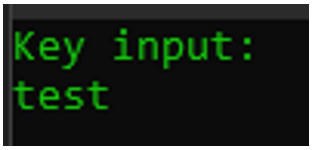
# VladMetz's CTF #5

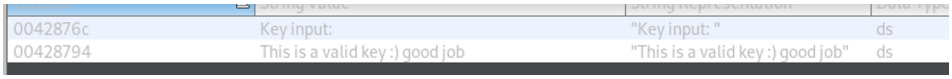
Friday, February 18, 2022 8:22 AM

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

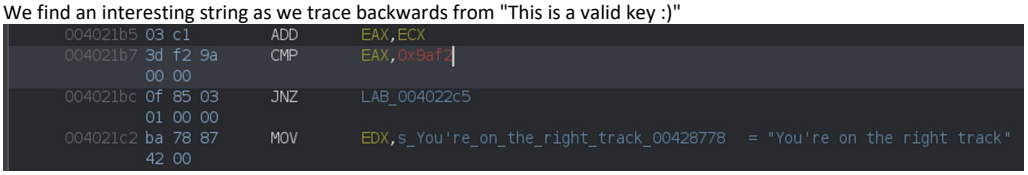



and when the wrong key is input the program exits

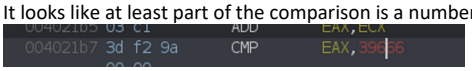
We look through the string in Ghidra to find a good starting point



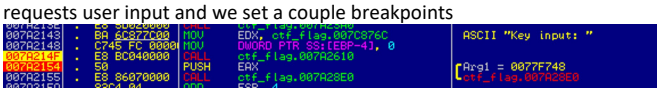

We find an interesting string as we trace backwards from "This is a valid key :)"



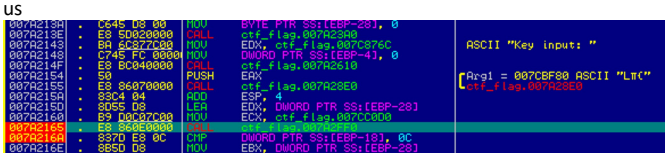

It looks like at least part of the comparison is a number....



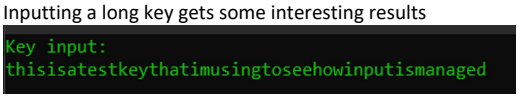

We run the program in ImmunityDebugger and find the point where the program requests user input and we set a couple breakpoints




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




Inputting a long key gets some interesting results



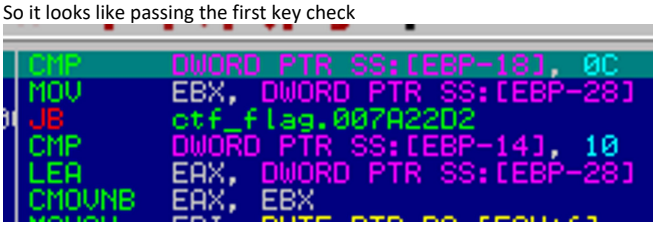


I manipulated registers to continue to the program to try to understand what was going on

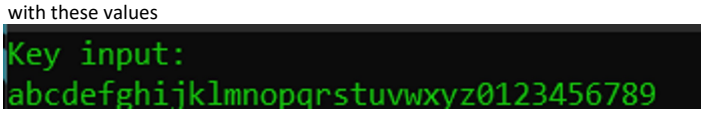
- LAHF instruction loads lower byte of the EFLAGS register into AH register.
- The lowest 8 bits of the flags are transferred:
  - Sign
  - Zero
  - Auxiliary Carry
  - Parity
  - Carry
- Instruction format:

```
lahf
```

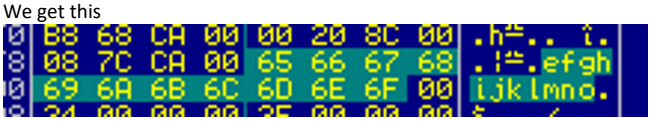
So it looks like passing the first key check



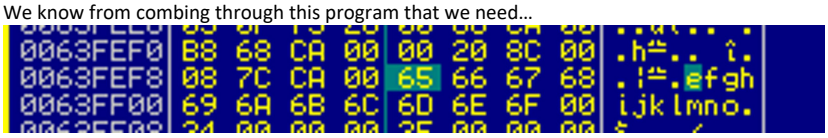

is a matter of setting certain values at certain points relative to ebp
with these values




We get this




We know from combing through this program that we need…




equal to 0x0C
and…


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         MOV     EBX, DWORD PTR SS:[EBP-28]
007A2171  .  0F82 5B010000   JB      ctf_flag.007A22D2
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
007A21B7  .  3D F29A0000     CMP     EAX, 9AF2
```

## The first check

```
.  E8 860E0000   CALL    ctf_flag.007A2FF0
.  837D E8 0C    CMP     DWORD PTR SS:[EBP-18], 0C
.  8B5D D8       MOV     EBX, DWORD PTR SS:[EBP-28]
.  0F82 5B01000( JB      ctf_flag.007A22D2
```

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/JNB 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/JNLE 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                  ctf_flag.007BE300
007A21B5  .  03C1            ADD     EAX, ECX
```

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** cmovnb *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, CMOVNB 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

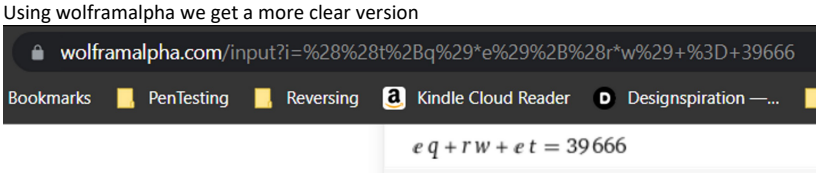
```
CMP     DWORD PTR SS:[EBP-14], 10
LEA     EAX, DWORD PTR SS:[EBP-28]
CMOVNB  EAX, EBX
MOVSX   EDI, BYTE PTR DS:[EAX+6]
LEA     EAX, DWORD PTR SS:[EBP-28]
CMOVNB  EAX, EBX
MOVSX   ESI, BYTE PTR DS:[EAX+7]
LEA     EAX, DWORD PTR SS:[EBP-28]
CMOVNB  EAX, EBX
MOVSX   EDX, BYTE PTR DS:[EAX+9]
LEA     EAX, DWORD PTR SS:[EBP-28]
CMOVNB  EAX, EBX
MOVSX   ECX, BYTE PTR DS:[EAX+A]
LEA     EAX, DWORD PTR SS:[EBP-28]
CMOVNB  EAX, 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 largets 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**

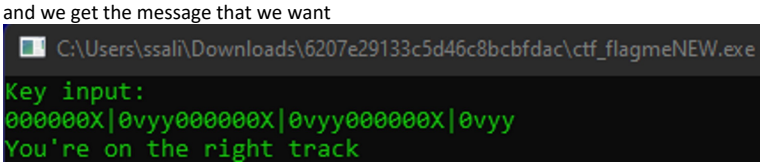
```
X|vyy
```

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

000000X|0vyy

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

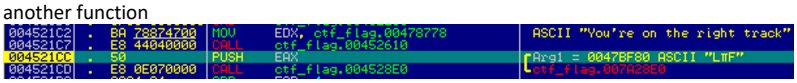

and we get the message that we want




Now onto the next series of checks

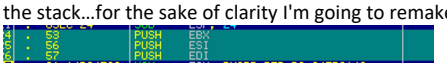
## The Second Check

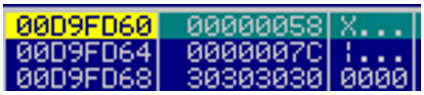
After we pass the arithmetic checks, our key gets passed to a function that prints and returns our input with some manipulations, this then gets passed onto another function




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

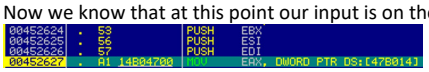


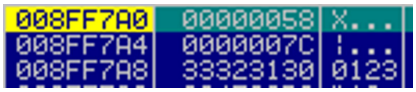




| 0th | 1st | 2nd | 3rd | 4th | 5th | 6th | 7th | 8th | 9th | 10th | 11th |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | X | \| | 8 | v | y | y |

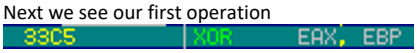
012345X|8vyy

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

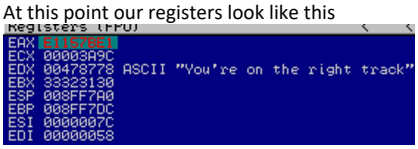



Next we see our first operation

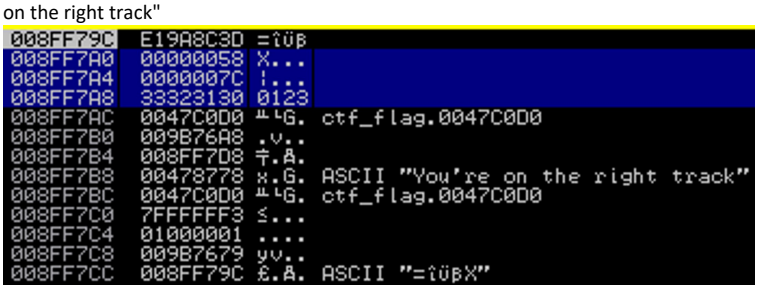


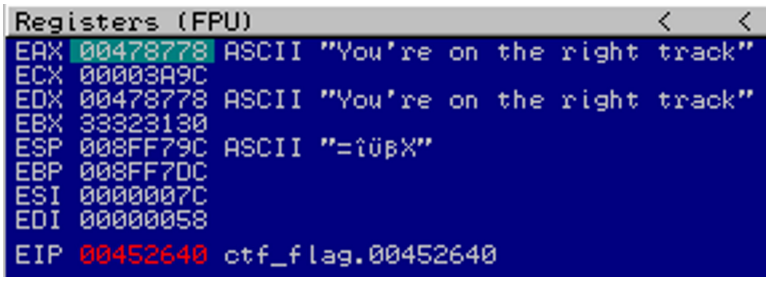

At this point our registers look like this




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

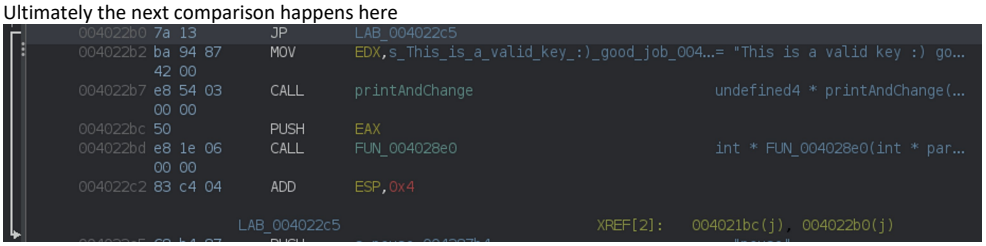
It looks like this XOR happens, then the program starts prepping to print "You're on the right track"



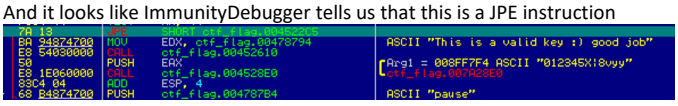




If we skip through the whole function until we return to main, then we end up with an arbitrary EAX getting passed into the next function


Skipping forward a few steps…it doesn't look like any of these checks matter. Ultimately the next comparison happens here

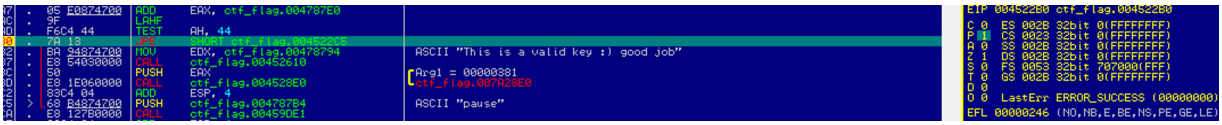


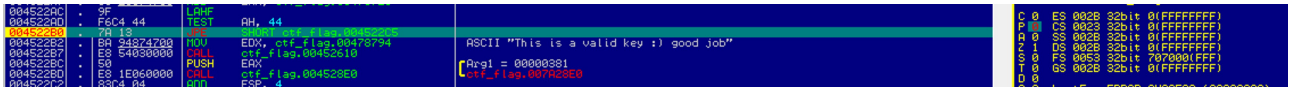

And it looks like ImmunityDebugger tells us that this is a JPE instruction




| | | |
|---|---|---|
| JP JPE | Jump if parity 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;

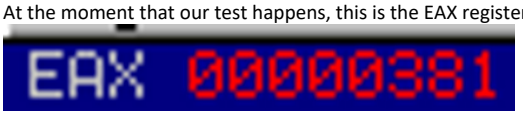
In conditional jumps, parity flag is used, where e.g. the JP instruction jumps to the given target when the parity flag is set and the JNP instruction jumps if it is not set. The flag may be also read directly with instructions such as PUSHF, which pushes the flags register on the stack.

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.[1] The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.

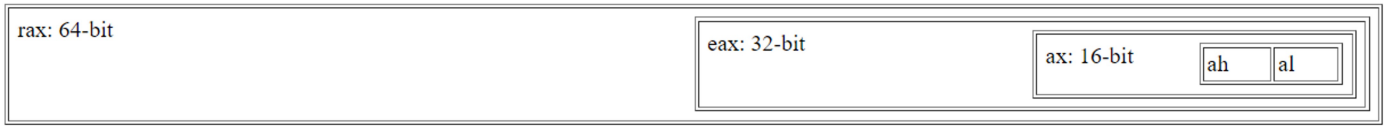
So it looks like the TEST before this comparison is being used to update the Parity Flag

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.[1]

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

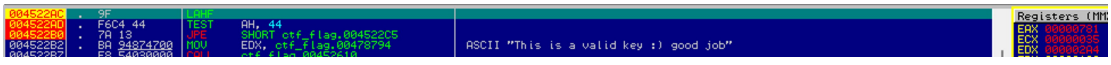

So we need to figure out how the program is setting the value of AH so that we can change our input to match

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
```

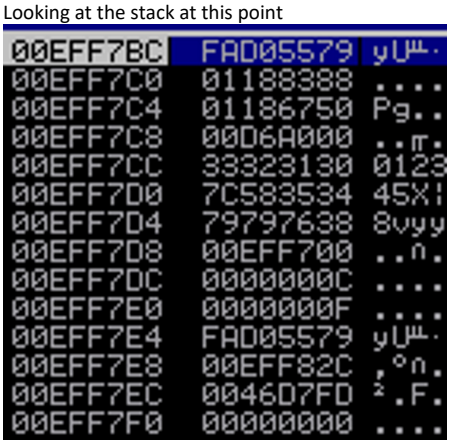
Before LAHF



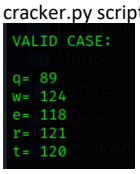

After LAHF


So it looks like we need to decrement the G flag

Looking at the stack at this point




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



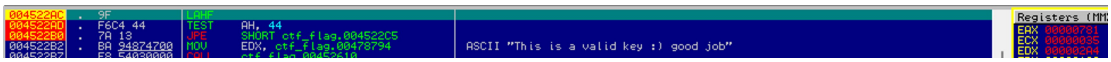

q= 89
w= 124
e= 118
r= 121
t= 120

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

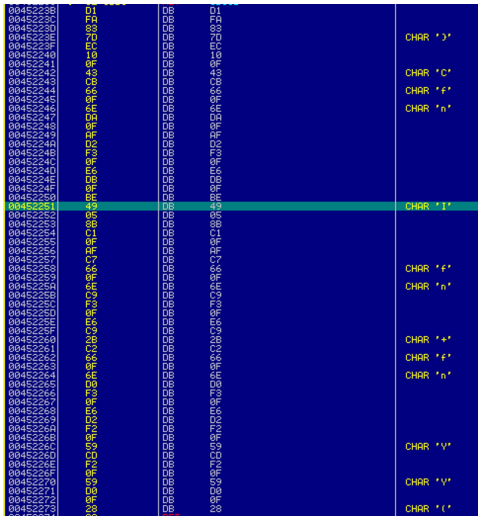
| 0th | 1st | 2nd | 3rd | 4th | 5th | 6th | 7th | 8th | 9th | 10th | 11th |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | Y | \| | 8 | v | y | x |

012345Y|8vyx

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

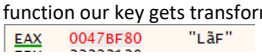



Since we know what values need to remain constant in order to get to this point, lets play around with some of the other values in order to try to drive a change in EAX



**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



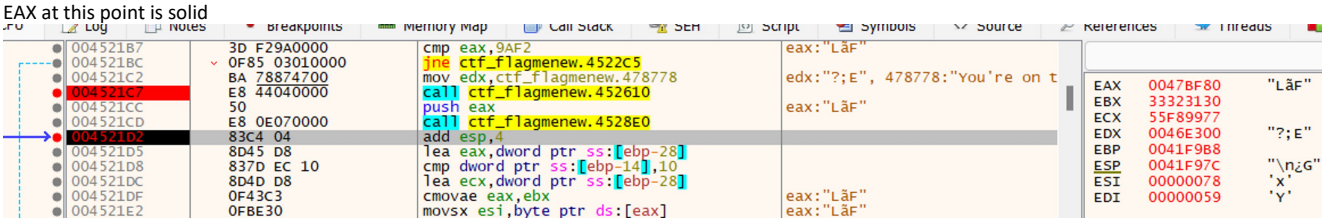

Key1:
012349Y|8vyx

Key2:
012345Yx8zr|

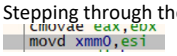
Key3:
092349Yx8zr|

Key 4:
012345YxAzr|

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




Stepping through the code we run into some interesting stuff



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 YMMx (x from 0 to 15) and can be used by the new AVX instructions, the lower half is XMMx, 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 user3096803

answered Jun 1, 2017 at 6:07 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 bits) 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 embedded 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.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

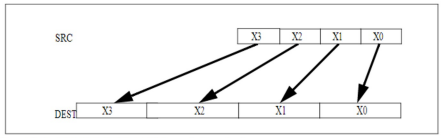



Figure 3-11. CVTDQ2PD (VEX.256 encoded version)

Straight forward enough, so we should expect the 0000…30 im xmm0 to become more significant bitwise

Original

```
XMM0  00000000000000000000000000000030
```

Results:

```
XMM0  00000000000000004048000000000000
XMM1  00000000000000000000000000000000
XMM2  00000000000000000000000000000000
XMM3  00000000000000000000000000000000
XMM4  00000000000000000000000000000000
XMM5  00000000000000000000000000000000
XMM6  00000000000000000000000000000000
XMM7  00000000000000000000000000000000

YMM0  00000000000000000000000000000000000000000000000004048000000000000
```

Interesting…next we move the 2nd value of our input into xmm5 and ymm5

```
XMM5  00000000000000000000000000000032
XMM6  00000000000000000000000000000000
XMM7  00000000000000000000000000000000

YMM0  00000000000000000000000000000000000000000000000004048000000000000
YMM1  00000000000000000000000000000000000000000000000000000000000000000
YMM2  00000000000000000000000000000000000000000000000000000000000000000
YMM3  00000000000000000000000000000000000000000000000000000000000000000
YMM4  00000000000000000000000000000000000000000000000000000000000000000
YMM5  00000000000000000000000000000000000000000000000000000000000000032
```

And then we have this instruction again

```
cvtdq2pd xmm5,xmm5
```

And this is the result

```
XMM5  00000000000000004049000000000000
XMM6  00000000000000000000000000000000
XMM7  00000000000000000000000000000000

YMM0  0000000000000000000000000000000000000000000000004048000000000000
YMM1  0000000000000000000000000000000000000000000000000000000000000000
YMM2  0000000000000000000000000000000000000000000000000000000000000000
YMM3  0000000000000000000000000000000000000000000000000000000000000000
YMM4  0000000000000000000000000000000000000000000000000000000000000000
YMM5  0000000000000000000000000000000000000000000000004049000000000000
```

Now things start to get very interesting

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

```
XMM5  00000000000000004039000000000000
XMM6  00000000000000000000000000000000
XMM7  00000000000000000000000000000000

YMM0  0000000000000000000000000000000000000000000000004048000000000000
YMM1  0000000000000000000000000000000000000000000000000000000000000000
YMM2  0000000000000000000000000000000000000000000000000000000000000000
YMM3  0000000000000000000000000000000000000000000000000000000000000000
YMM4  0000000000000000000000000000000000000000000000000000000000000000
YMM5  0000000000000000000000000000000000000000000000004039000000000000
```

Then we see this

```
XMM0  00000000000000004048000000000000
XMM1  00000000000000000000000000000000
XMM2  00000000000000000000000000000000
XMM3  00000000000000000000000000000000
XMM4  00000000000000004039000000000000
XMM5  00000000000000004039000000000000
XMM6  00000000000000000000000000000000
XMM7  00000000000000000000000000000000

YMM0  0000000000000000000000000000000000000000000000004048000000000000
YMM1  0000000000000000000000000000000000000000000000000000000000000000
YMM2  0000000000000000000000000000000000000000000000000000000000000000
YMM3  0000000000000000000000000000000000000000000000000000000000000000
YMM4  0000000000000000000000000000000000000000000000004039000000000000
YMM5  0000000000000000000000000000000000000000000000004039000000000000
YMM6  0000000000000000000000000000000000000000000000000000000000000000
YMM7  0000000000000000000000000000000000000000000000000000000000000000
```

Then

```
XMM0  00000000000000004048000000000000
XMM1  00000000000000000000000000000000
XMM2  00000000000000000000000000000000
XMM3  0000000000000000000000000000001A
XMM4  00000000000000004039000000000000
XMM5  00000000000000004039000000000000
XMM6  00000000000000000000000000000000
XMM7  00000000000000000000000000000000

YMM0  0000000000000000000000000000000000000000000000004048000000000000
YMM1  0000000000000000000000000000000000000000000000000000000000000000
YMM2  0000000000000000000000000000000000000000000000000000000000000000
YMM3  000000000000000000000000000000000000000000000000000000000000001A
YMM4  0000000000000000000000000000000000000000000000004039000000000000
YMM5  0000000000000000000000000000000000000000000000004039000000000000
YMM6  0000000000000000000000000000000000000000000000000000000000000000
YMM7  0000000000000000000000000000000000000000000000000000000000000000
```

Then

```
XMM0  00000000000000004048000000000000
XMM1  00000000000000000000000000000000
XMM2  00000000000000000000000000000000
XMM3  0000000000000000403A000000000000
XMM4  00000000000000004039000000000000
XMM5  00000000000000004039000000000000
XMM6  00000000000000000000000000000000
XMM7  00000000000000000000000000000000

YMM0  0000000000000000000000000000000000000000000000004048000000000000
YMM1  0000000000000000000000000000000000000000000000000000000000000000
YMM2  0000000000000000000000000000000000000000000000000000000000000000
YMM3  000000000000000000000000000000000000000000000000403A000000000000
YMM4  0000000000000000000000000000000000000000000000004039000000000000
YMM5  0000000000000000000000000000000000000000000000004039000000000000
YMM6  0000000000000000000000000000000000000000000000000000000000000000
YMM7  0000000000000000000000000000000000000000000000000000000000000000
```

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          cmovae eax,ebx
0045222E    F3:0FE6E4       cvtdq2pd xmm4,xmm4
00452232    0FBE40 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          cmovae ecx,ebx
00452244    66:0F6EDA       movd xmm3,edx
00452248    0FAFD2          imul edx,edx
0045224B    F3:0FE6DB       cvtdq2pd xmm3,xmm3
0045224F    0FBE49 05       movsx ecx,byte ptr ds:[ecx+5]
00452253    8BC1            mov eax,ecx
EIP -> 00452255    0FAFC7          imul eax,edi
00452258    66:0F6EC9       movd xmm1,ecx
0045225C    F3:0FE6C9       cvtdq2pd xmm1,xmm1

EAX   00000035   '5'
EBX   33323130
ECX   00000035   '5'
EDX   000002A4   L'¤'
EBP   00CFF9FC   &"TúÏ"
ESP   00CFF9C4
ESI   00000030   '0'
EDI   00000031   '1'

EIP   00452255   ctf_flagmenew.00452255

EFLAGS   00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1
```

Result

```
EAX   00000A25   L'¤'
EBX   33323130
```

Final load out of the registers before any arithmetic operations

```
XMM0  00000000000000004048000000000000
XMM1  0000000000000000404A800000000000
XMM2  00000000000000004090E040000000000
XMM3  0000000000000000403A000000000000
XMM4  00000000000000004039000000000000
XMM5  00000000000000004039000000000000
XMM6  00000000000000000000000000000000
XMM7  00000000000000000000000000000000

YMM0  0000000000000000000000000000000000000000000000004048000000000000
YMM1  0000000000000000000000000000000000000000000000000404A800000000000
YMM2  00000000000000000000000000000000000000000000000004090E040000000000
YMM3  0000000000000000000000000000000000000000000000000403A000000000000
YMM4  0000000000000000000000000000000000000000000000000039000000000000
YMM5  0000000000000000000000000000000000000000000000000039000000000000
YMM6  0000000000000000000000000000000000000000000000000000000000000000
YMM7  0000000000000000000000000000000000000000000000000000000000000000
```

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 EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

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

Before ucomiss

```
EFLAGS   00000304
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1
```

After ucomiss

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

Before LAHF

```
EAX   00000781   L'ゃ'
EBX   33323130
ECX   00000035   '5'
EDX   000002A4   L'¤'
EBP   007EF9A0   &"øù~"
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

```
EAX   00000381
EBX   33323130
ECX   00000035   '5'
EDX   000002A4   L'¤'
EBP   007EF9A0   &"øù~"
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   00BFF9C4   "012345Yx8zr|"
EDX   00000034   '4'
EBP   00BFF9EC   &"Dú¿"
ESP   00BFF9B4
ESI   00000030   '0'
EDI   00000031   '1'

EIP   0045223B   ctf_flagmenew.0045223B
```

After

```
EAX   00000034   '4'
EBX   33323130
ECX   00BFF9C4   "012345Yx8zr|"
EDX   0000001A
EBP   00BFF9EC   &"Dú¿"
ESP   00BFF9B4
ESI   00000030   '0'
EDI   00000031   '1'
```

Next we multiply edx by itself

```
imul edx,edx
```

Result

```
EDX   000002A4   L'¤'
```

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

```
imul eax,edi
```

```
EAX   00000A25   L'¥'
EBX   33323130
ECX   00000035   '5'
EDX   000002A4   L'¤'
EBP   00BFF9EC   &"Dú¿"
ESP   00BFF9B4
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
ECX   00000035   '5'
EDX   000002A4   L'¤'
EBP   00BFF9EC   &"Dú¿"
ESP   00BFF9B4
ESI   00000030   '0'
EDI   00000031   '1'

EIP   00452260   ctf_flagmenew.00452260
```

```
EAX   00000781   L'ˇ'
EBX   33323130
ECX   00000035   '5'
EDX   000002A4   L'¤'
EBP   00BFF9EC   &"Dú¿"
ESP   00BFF9B4
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 mutliplcation because that's what ends up getting loaded into AH and therefore the partiy 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 | \| | 8 | v | y | x |

~~~~~X|8vyy
Key to our success is the following instructions:
~~~~~

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

And this value

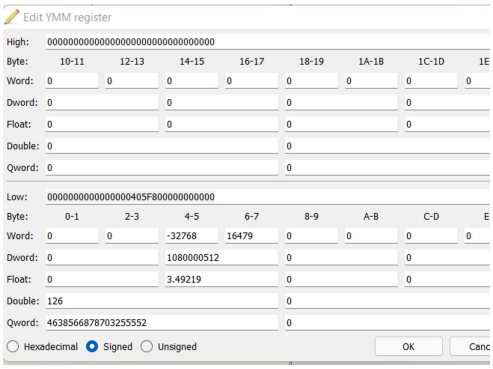
| Address | Hex |
|---|---|
| 004787E0 | AC 2C 33 49 |
| 004787E0 | 02 00 00 00 |

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




There IS a bunch of arithmetic we need to keep track of so let's break that down
Given the key:
012345X|8vyy

| | 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 | | | | | | |

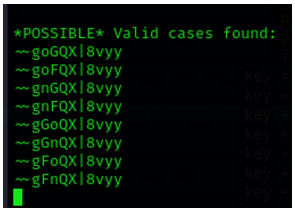
At the start---
xmm0 - 48 (0th term)
xmm1 - 53 (5th term)
xmm2 - 1921 ((multiply 5th and 1st term)-(4th term shift right squared))
xmm3 - 26 (fourth term, arithmetic shift right 1)
xmm4 - 25 (third term, arithmetic shift right 1)
xmm5 - 25 (second term /2)

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|8vyy'
                    print(key
```




The keys I tested look like:
~~goGQX|8vyy

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