

For Realz Mode!

Reversing an MBR from the CSAW CTF
(and how to write a keygen with a SAT solver)

t0x0 numinit


DC858, DC619, and V A P O R S E C

2017-10-11

- Morgan (@numinit)
 - Software developer by day
 - Tinkerer on everything by night
 - First Def Con and Toorcon experiences this year
 - Doing CSAW CTF for a while
- t0x0 (@t0x0pg)
 - Writes lots of interpreted code
 - Lives mostly in Windows world
 - Jack of all trades, master of none (so far)
 - Obsessively curious

What's CSAW?

An annual security capture the flag run by New York Polytechnic that contains challenges with a **wide range of difficulties**, attracting everyone from undergraduates to well-known teams

CSAW'17  CAPTURE THE FLAG		
Crypto baby_crypt 350	Reversing Gopherz 350	Pwn FIREWALL 400
Reversing bananaScript 450	Pwn Minesweeper 500	Reversing GrumpCheck 500

What's a security capture the flag?

- 48 hours of caffeinated reverse engineering and exploitation

What's a security capture the flag?

- 48 hours of caffeinated reverse engineering and exploitation
- More seriously: an event where you break a bunch of programs to retrieve hidden strings of text (the flags)

What's a security capture the flag?

- 48 hours of caffeinated reverse engineering and exploitation
- More seriously: an event where you break a bunch of programs to retrieve hidden strings of text (the flags)
- An opportunity to learn new things - we wouldn't even be presenting if it wasn't for this CTF

Who are V A P O R S E C ?

- A new A E S T H E T I C San Diego CTF team
- Had 10 participants during the CSAW CTF
- Achieved 15th in CSAW industry professional bracket, not bad for the first time

RE400 what?...

- Challenge worth 400 points
- Reverse Engineering category
- We get some hints right away...
 - This is an MBR
 - ...from an x86 system

Reversing

101 Solves

realism

Did you know that x86 is really old? I found a really old Master Boot Record that I thought was quite interesting! At least, I think it's really old...

```
qemu-system-i386 -drive format=raw,file=main.bin
```

main.bin

Key

SUBMIT

A place to start...

- Wikipedia, of course!



The screenshot shows the Wikipedia page for 'Master boot record'. The left sidebar contains navigation links such as 'Main page', 'Contents', 'Featured content', 'Current events', 'Random article', 'Donate to Wikipedia', and 'Wikipedia store'. The main content area has tabs for 'Article' and 'Talk'. The article title 'Master boot record' is prominently displayed, followed by the text 'From Wikipedia, the free encyclopedia'. The article body begins with a note: 'This article is about a PC-specific type of boot sector c'. The first paragraph defines a **master boot record (MBR)** as a special type of boot sector intended for use with IBM PC-compatible systems and beyond. It explains that the MBR holds information on how the logical partition function as a loader for the installed operating system—using a **boot record (VBR)**. This MBR code is usually referred to as a **boot record (VBR)**. The second paragraph discusses the organization of the partition table in the MBR, noting that it limits the limit assuming 33-bit arithmetics or 4096-byte sectors are operating systems and system tools, and can cause serious partitioning scheme is in the process of being superseded provide some limited form of backward compatibility for older MBRs are not present on non-partitioned media such as floppy disks.

Below the text, there is a 'Contents' section with a '[hide]' link. The first item in the table of contents is '1 Overview'.

A place to start...

- Wikipedia, of course!
 - 512 bytes
 - MBR signature: 55 AA
 - "expected to contain real mode machine language instructions"
 - little-endian
 - loads at 0000:7C00



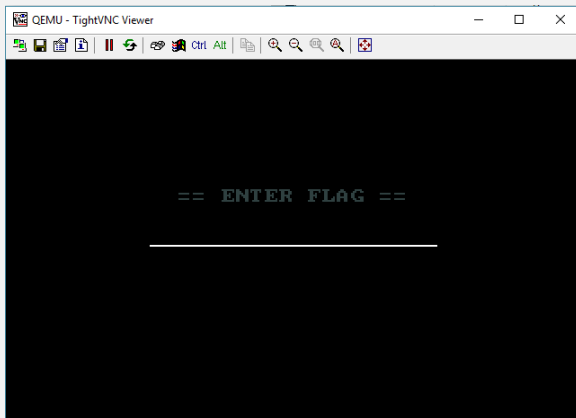
The screenshot shows the Wikipedia article for "Master boot record". At the top is the Wikipedia logo, a globe made of puzzle pieces with various symbols. Below it is the text "WIKIPEDIA The Free Encyclopedia". To the left of the main content is a sidebar with links: "Main page", "Contents", "Featured content", "Current events", "Random article", "Donate to Wikipedia", "Wikipedia store", "Interaction", "Help", "About Wikipedia", "Community portal", "Recent changes", "Contact page", "Tools", "What links here", and "Related changes". The main content area has a tabbed interface with "Article" and "Talk" tabs. The title "Master boot record" is prominently displayed. Below the title is the text "From Wikipedia, the free encyclopedia". The article body starts with a sentence: "This article is about a PC-specific type of *boot sector* c". Below this is a paragraph: "A **master boot record (MBR)** is a special type of *boot sector* intended for use with *IBM PC-compatible* systems and bey". Another paragraph follows: "The MBR holds the information on how the logical partition function as a loader for the installed operating system—us". A third paragraph: "*record* (VBR). This MBR code is usually referred to as a b". A fourth paragraph: "The organization of the partition table in the MBR limits the limit assuming 33-bit arithmetics or 4096-byte sectors are operating systems and system tools, and can cause serio". A fifth paragraph: "partitioning scheme is in the process of being superseded provide some limited form of backward compatibility for ok". A sixth paragraph: "MBRs are not present on non-partitioned media such as fl". At the bottom of the article is a "Contents" section with a "[hide]" link. The first item in the table of contents is "1 Overview".

Tool Time!...

qemu (gift wrapped)

- -s (gdb)
- -S (suspend)
- -vnc:1

```
qemu-system-i386 -s -S -vnc :1 -drive format=raw,file=main.bin
```



Tool Time!...

qemu (gift wrapped)

- QEMU/Monitor
 - info registers
 - system reset

```
ES =9f40 0009f400 0000ffff 00009300
CS =f000 000f0000 0000ffff 00009b00
SS =0000 00000000 0000ffff 00009300
DS =0000 00000000 0000ffff 00009300
FS =0000 00000000 0000ffff 00009300
GS =0000 00000000 0000ffff 00009300
LDT=0000 00000000 0000ffff 00008200
TR =0000 00000000 0000ffff 00008b00
GDT= 000fd3a8 00000037
IDT= 00000000 0000003f
CR0=00000012 CR2=00000000 CR3=00000000 CR4=00000600
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0fff DR7=00000400
EFER=0000000000000000
FCW=037f FSW=0000 IST=01 FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
(qemu) █
```

Tool Time!...

gdb

- target remote localhost:1234
- set architecture i8086 (bootloaders are 16 bit, right?)
- display/i \$pc - print program counter
- br *0xADDR - set breakpoint
- si - run one instruction
- c - continue

```
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) target remote localhost:1234
```

```
(gdb) target remote localhost:1234
```

Tool Time!...

gdb

- info reg

```
(gdb) info reg
eax                0x0          0
ecx                0x8b          139
edx                0x26183       156035
ebx                0x6ef2       28402
esp                0x6ed2       0x6ed2
ebp                0x1234       0x1234
esi                0x0          0
edi                0x0          0
eip                0xbdd2       0xbdd2
eflags             0x46          [ PF ZF ]
cs                 0xf000       61440
ss                 0x0          0
ds                 0x0          0
es                 0x9f40       40768
fs                 0x0          0
gs                 0x0          0
(gdb)
```

Tool Time!...

gdb

- info reg
- info frame

```
(gdb) info frame
Stack level 0, frame at 0x123c:
    eip = 0x656a; saved eip 0x5f5f5f5f
    called by frame at 0x5f5f5f49
    Arglist at 0x1234, args:
    Locals at 0x1234, Previous frame's sp is 0x123c
    Saved registers:
        ebp at 0x1234, eip at 0x1238
```


Tool Time!...

gdb

- info reg
- info frame
- x /CT 0xADDR - display C units of T type from ADDR
- set {int}0xADDR = 42
- set {char[4]} 0xADDR = "AAA"

```
(gdb) x /8x 0x1234
0x1234: 0x5f 0x5f 0x5f 0x5f 0x5f 0x5f 0x5f 0x5f
(gdb) set {int} 0x1234 = 41
(gdb) x /8x 0x1234
0x1234: 0x29 0x00 0x00 0x00 0x5f 0x5f 0x5f 0x5f
(gdb) set {int} 0x1234 = 0x41
(gdb) x /8x 0x1234
0x1234: 0x41 0x00 0x00 0x00 0x5f 0x5f 0x5f 0x5f
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
```

Tool Time!...

gdb

- info reg
- info frame
- x /CT 0xADDR - display C units of T type from ADDR
- set {int}0xADDR = 42
- set {char[4]} 0xADDR = "AAA"

```
(gdb) x /8x 0x1234
0x1234: 0x5f 0x5f 0x5f 0x5f 0x5f 0x5f 0x5f 0x5f
(gdb) set {int} 0x1234 = 41
(gdb) x /8x 0x1234
0x1234: 0x29 0x00 0x00 0x00 0x5f 0x5f 0x5f 0x5f
(gdb) set {int} 0x1234 = 0x41
(gdb) x /8x 0x1234
0x1234: 0x41 0x00 0x00 0x00 0x5f 0x5f 0x5f 0x5f
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.

(gdb) x /8x 0x1234
0x1234: 0x5f 0x5f 0x5f 0x41 0x5f 0x5f 0x5f 0x5f
(gdb) set {int} 0x1234 = 0x5f5f5f41
(gdb) x /8x 0x1234
0x1234: 0x41 0x5f 0x5f 0x5f 0x5f 0x5f 0x5f 0x5f
```

Tool Time!...

gdb

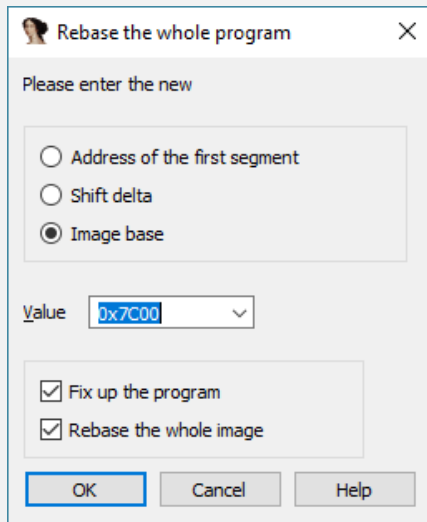
- info reg
- info frame
- x /CT 0xADDR - display C units of T type from ADDR
- set {int}0xADDR = 42
- set {char[4]} 0xADDR = "AAA"



Tool Time!...

IDA

■ Rebase

 **Rebase the whole program** ✕

Please enter the new

☐ Address of the first segment

☐ Shift delta

☒ Image base

Value

☒ Fix up the program

☒ Rebase the whole image

Tool Time!...

IDA

- Rebase
- Common ASM
 - int
 - mov
 - inc, dec
 - and, or, not, xor...
 - cmp
 - jmp, jz, jge, jle...

```
seg000:7C00      mov     ax, 13h
seg000:7C03      int      10h                ; - VIDEO - SET VIDEO MODE
seg000:7C03                        ; AL = mode
seg000:7C05      mov     eax, cr0
seg000:7C08      and     ax, 0FFFBh
seg000:7C0B      or      ax, 2
seg000:7C0E      mov     cr0, eax
seg000:7C11      mov     eax, cr4
seg000:7C14      or      ax, 600h
seg000:7C17      mov     cr4, eax
seg000:7C1A      mov     word ptr ds:1266h, 0Ah
seg000:7C20      mov     bx, 0
```

Tool Time!...

IDA

- Rebase
- Common ASM
- Registers
 - E AX - Accumulator
 - E CX - Counter
 - E DX - Data
 - E BX - Base
 - E SP - Stack pointer
 - E BP - Stack base pointer
 - E SI - Source
 - E DI - Destination

```
seg000:7C00    mov     ax, 13h
seg000:7C03    int     10h           ; - VIDEO - SET VIDEO MODE
seg000:7C03                                ; AL = mode
seg000:7C05    mov     eax, cr0
seg000:7C08    and     ax, 0FFFBh
seg000:7C0B    or      ax, 2
seg000:7C0E    mov     cr0, eax
seg000:7C11    mov     eax, cr4
seg000:7C14    or      ax, 600h
seg000:7C17    mov     cr4, eax
seg000:7C1A    mov     word ptr ds:1266h, 0Ah
seg000:7C20    mov     bx, 0
```

- Look for hints

Disassembly...

- Look for hints
 - Loops

```
seg000:7C23 loc_7C23:          ; CODE XREF: seg000:7C2C4j
mov     byte ptr [bx+129Ah], 5Fh ; '_'
inc     bx
seg000:7C29          ; CODE XREF: seg000:7C2C4j
cmp     bx, 15h
jle     short loc_7C23
seg000:7C2C          ; CODE XREF: seg000:7C2C4j
mov     ds:byte_7D63+65h, 0

loc_7C29:
mov     byte ptr [bx+129Ah], 5Fh ; '_'
inc     bx
cmp     bx, 15h
jle     short loc_7C23
mov     ds:byte_7D63+65h, 0
```


- Look for hints
 - Loops
 - Comparisons

```
seg000:7C6F      cmp     dword ptr ds:1234h, 67616C66h  
seg000:7C78      jnz     loc_7D4D
```

- Look for hints
 - Loops
 - Comparisons
 - Unknowns

```
seg000:7C7C      movaps   xmm0, xmmword ptr ds:1238h
seg000:7C81      movaps   xmm5, xmmword ptr ds:loc_7C00
seg000:7C86      pshufd   xmm0, xmm0, 1Eh
seg000:7C8B      mov      si, 8
seg000:7C8E      loc_7C8E:                                ; CODE XREF: seg000:7CC1↓j
seg000:7C8E      movaps   xmm2, xmm0
seg000:7C91      andps    xmm2, xmmword ptr [si+7D90h]
seg000:7C96      psadbw   xmm5, xmm2
seg000:7C9A      movaps   xmmword ptr ds:1268h, xmm5
```

Putting it all together...

- We know that 0x7C6F compares user input to "flag"

```
seg000:7C6F      cmp     dword ptr ds:1234h, 67616C66h  
seg000:7C78      jnz     loc_7D4D
```

Putting it all together...

- We know that 0x7C6F compares user input to "flag"
- There's a bunch of complicated instructions right after that cmp

```
seg000:7C7C      movaps   xmm0, xmmword ptr ds:1238h
seg000:7C81      movaps   xmm5, xmmword ptr ds:loc_7C80
seg000:7C86      pshufd   xmm0, xmm0, 1Eh
seg000:7C8B      mov     si, 8
seg000:7C8E      loc_7C8E: ; CODE XREF: seg000:7C81↓j
seg000:7C8E      movaps   xmm2, xmm0
seg000:7C8E      andps    xmm2, xmmword ptr [si+7D90h]
seg000:7C91      psadbw   xmm5, xmm2
seg000:7C96      movaps   xmmword ptr ds:1268h, xmm5
seg000:7C9A
```

Putting it all together...

- We know that 0x7C6F compares user input to "flag"
- There's a bunch of complicated instructions right after that cmp
- We can use gdb and qemu/monitor to see what's happening...

```
XMM00=7d41414141414141414141414141417b XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=220f02c883fbe083c0200f10cd0013b8
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
```

Putting it all together...

- We know that 0x7C6F compares user input to "flag"
- There's a bunch of complicated instructions right after that cmp
- We can use gdb and qemu/monitor to see what's happening...
- Now we just have to work backwards from there.

```
XMM00=7d41414141414141414141414141417b XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=220f02c883fbe083c0200f10cd0013b8
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
```

```
XMM00=4141417b414141414141414141414141 XMM01=00000000000000000000000000000000
XMM02=4141417b414141007d41414141414100 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=220f02c883fbe083c0200f10cd0013b8
```

Other useful tools...that I didn't know about

- Binary Ninja
- pwndbg
- Radare

A quick recap...

- We have an x86 boot sector that's asking us for the flag



== ENTER FLAG ==

A quick recap...

- We have an x86 boot sector that's asking us for the flag
- The flag is clearly in the boot sector SOMEWHERE...



== ENTER FLAG ==

A quick recap...

- We have an x86 boot sector that's asking us for the flag
- The flag is clearly in the boot sector SOMEWHERE...
 - ...but not in plaintext, because this is a 400 point challenge and that would be too easy



== ENTER FLAG ==

A quick recap...

- We have an x86 boot sector that's asking us for the flag
- The flag is clearly in the boot sector SOMEWHERE...
 - ...but not in plaintext, because this is a 400 point challenge and that would be too easy
- So, where is it?



== ENTER FLAG ==

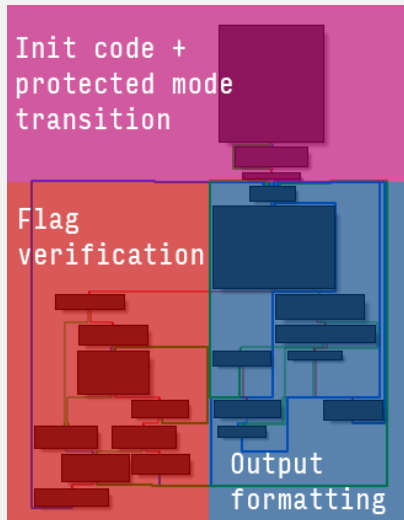
Reversing the boot sector

- Open the file in your favorite disassembler (e.g. IDA); rebase at 0x7c00



Reversing the boot sector

- Open the file in your favorite disassembler (e.g. IDA); rebase at 0x7c00
- We can visally pick out three sections:
 - Init code (responsible for the protected mode transition)
 - Display code (identifiable by a large number of `int` instructions)
 - Some other code that uses Intel SSE2 instructions



First leads

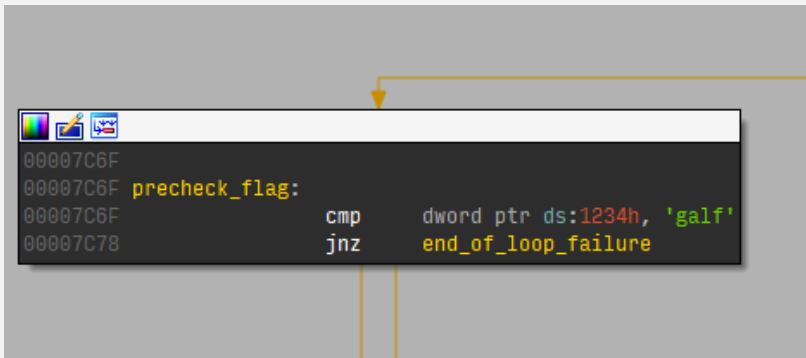
- The program asks you to enter 20 characters, and immediately breaks out if the first 4 characters aren't 'flag' after you enter character #20

First leads

- The program asks you to enter 20 characters, and immediately breaks out if the first 4 characters aren't 'flag' after you enter character #20
- We can verify this in a debugger after eyeballing the code

First leads

- The program asks you to enter 20 characters, and immediately breaks out if the first 4 characters aren't 'flag' after you enter character #20
- We can verify this in a debugger after eyeballing the code



As it turns out...

After several hours of staring at x86 assembly

- The flag is **hashed** using a **custom algorithm** implemented with Intel SSE instructions (and this isn't very surprising)

As it turns out...

After several hours of staring at x86 assembly

- The flag is **hashed** using a **custom algorithm** implemented with Intel SSE instructions (and this isn't very surprising)
- *Ugh...*

As it turns out...

After several hours of staring at x86 assembly

- The flag is **hashed** using a **custom algorithm** implemented with Intel SSE instructions (and this isn't very surprising)
- *Ugh...*
- We have to find an input to the hash algorithm that hashes to the same value that's stored in the boot sector

Suddenly, SSE2 instructions

“My Intel CPU can do *that?*”

The author of the challenge decided to use a bunch of obscure x86 SSE2 instructions to force us to trawl through Intel documentation

Note

SSE2 instructions operate on XMM registers, which are 128 bits (16 bytes) wide.

Suddenly, SSE2 instructions

“My Intel CPU can do *that?*”

The author of the challenge decided to use a bunch of obscure x86 SSE2 instructions to force us to trawl through Intel documentation

- movaps
- andps
- pshufd
- psadbw

Note

SSE2 instructions operate on XMM registers, which are 128 bits (16 bytes) wide.

Suddenly, SSE2 instructions

“My Intel CPU can do *that*?”

The author of the challenge decided to use a bunch of obscure x86 SSE2 instructions to force us to trawl through Intel documentation

- movaps: Moves to/from/between XMM registers
- andps: Performs bitwise AND between XMM registers
- pshufd: Reorders 32-bit words in XMM registers
- psadbw: Sums absolute values of differences between bytes (it's as crazy as it sounds)

Note

SSE2 instructions operate on XMM registers, which are 128 bits (16 bytes) wide.

Suddenly, SSE2 instructions

“My Intel CPU can do *that*?”

The author of the challenge decided to use a bunch of obscure x86 SSE2 instructions to force us to trawl through Intel documentation

- `movaps`: Moves to/from/between XMM registers
- `andps`: Performs bitwise AND between XMM registers
- `pshufd`: Reorders 32-bit words in XMM registers
- `psadbw`: Sums absolute values of differences between bytes (it's as crazy as it sounds)

Note

SSE2 instructions operate on XMM registers, which are 128 bits (16 bytes) wide.

Question

Why are these instructions useful for writing a hash function (even if it's a bad hash function that we can break?)

Packed Sum of Absolute Differences of Bytes in Word

That's a mouthful...

xmm0:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

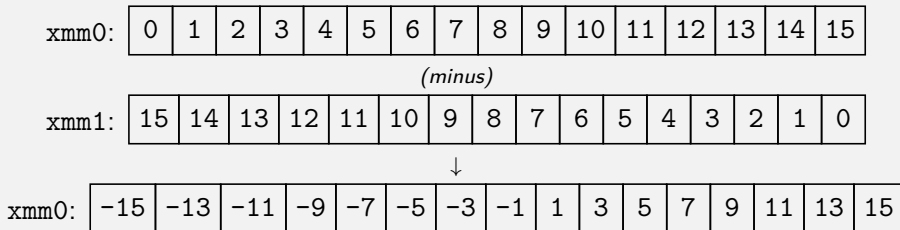
Packed Sum of Absolute Differences of Bytes in Word

That's a mouthful...



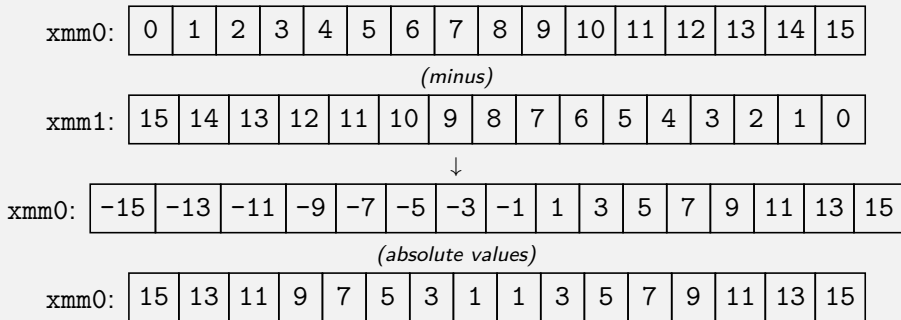
Packed Sum of Absolute Differences of Bytes in Word

That's a mouthful...



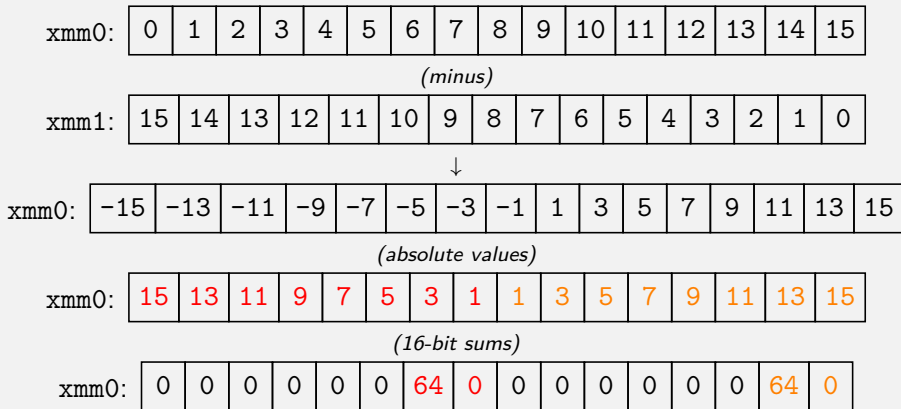
Packed Sum of Absolute Differences of Bytes in Word

That's a mouthful...



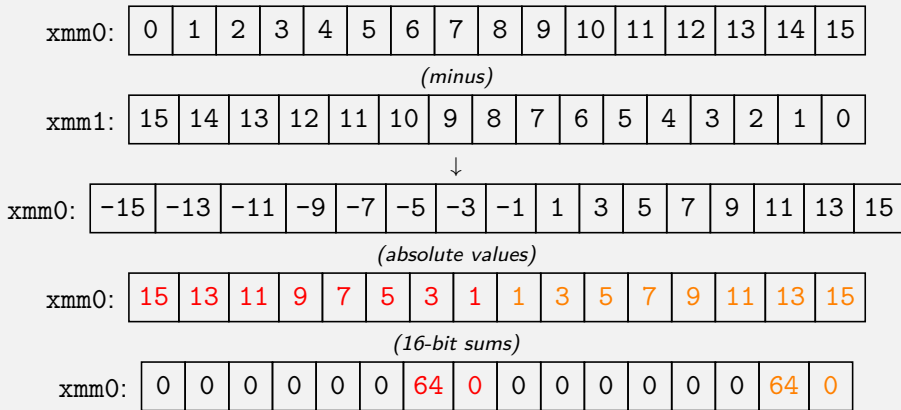
Packed Sum of Absolute Differences of Bytes in Word

That's a mouthful...



Packed Sum of Absolute Differences of Bytes in Word

That's a mouthful...



It's hard to go back...

You have to find two sets of 8 bytes where the absolute values of their differences sum to 64

SAT solvers to the rescue!

How to master Sudoku without memorizing strategies

“I think a serious case can be made that the decline in the American economy can be blamed on the sapping of the mental energy and productivity of the American workforce that sudoku addiction alone has wrought”

– Some Slate writer

SAT solvers to the rescue!

How to master Sudoku without memorizing strategies

"I think a serious case can be made that the decline in the American economy can be blamed on the sapping of the mental energy and productivity of the American workforce that sudoku addiction alone has wrought"

– Some Slate writer

Notice anything strange about this puzzle?

d	c	6	1	9	0	e	b	5	7	3	2	4	8	f	a
5	8	a	4	f	c	2	d	9	0	b	e	3	7	6	1
2	b	9	7	5	3	6	1	4	8	a	f	d	0	e	c
0	3	f	e	4	a	8	7	d	c	6	1	9	b	2	5
8	d	c	6	1	9	3	2	b	5	e	7	0	4	a	f
b	1	5	0	d	8	c	6	f	a	9	4	7	2	3	e
9	a	4	f	7	5	0	e	2	3	1	8	b	d	c	6
7	2	e	3	b	4	a	f	0	d	c	6	1	9	5	8
e	4	d	c	6	1	9	0	a	b	2	5	8	f	7	3
1	9	8	2	3	7	5	4	e	6	f	0	a	c	d	b
f	6	7	a	2	d	b	c	1	9	8	3	5	e	0	4
3	5	0	b	a	e	f	8	7	4	d	c	6	1	9	2
a	e	3	d	c	6	1	9	8	f	7	b	2	5	4	0
6	0	2	5	e	b	d	3	c	1	4	9	f	a	8	7
c	7	1	9	8	f	4	5	6	2	0	a	e	3	b	d
4	f	b	8	0	2	7	a	3	e	5	d	c	6	1	9

SAT solvers to the rescue!

How to master Sudoku without memorizing strategies

"I think a serious case can be made that the decline in the American economy can be blamed on the sapping of the mental energy and productivity of the American workforce that sudoku addiction alone has wrought"

– Some Slate writer

Notice anything strange about this puzzle?
Stranger still: a SAT solver created it from thin air

d	c	6	1	9	0	e	b	5	7	3	2	4	8	f	a
5	8	a	4	f	c	2	d	9	0	b	e	3	7	6	1
2	b	9	7	5	3	6	1	4	8	a	f	d	0	e	c
0	3	f	e	4	a	8	7	d	c	6	1	9	b	2	5
8	d	c	6	1	9	3	2	b	5	e	7	0	4	a	f
b	1	5	0	d	8	c	6	f	a	9	4	7	2	3	e
9	a	4	f	7	5	0	e	2	3	1	8	b	d	c	6
7	2	e	3	b	4	a	f	0	d	c	6	1	9	5	8
e	4	d	c	6	1	9	0	a	b	2	5	8	f	7	3
1	9	8	2	3	7	5	4	e	6	f	0	a	c	d	b
f	6	7	a	2	d	b	c	1	9	8	3	5	e	0	4
3	5	0	b	a	e	f	8	7	4	d	c	6	1	9	2
a	e	3	d	c	6	1	9	8	f	7	b	2	5	4	0
6	0	2	5	e	b	d	3	c	1	4	9	f	a	8	7
c	7	1	9	8	f	4	5	6	2	0	a	e	3	b	d
4	f	b	8	0	2	7	a	3	e	5	d	c	6	1	9

What's a SAT (or SMT) solver?

- SAT solvers

What's a SAT (or SMT) solver?

- SAT solvers
 - Only operate on Boolean logic

What's a SAT (or SMT) solver?

- SAT solvers

- Only operate on Boolean logic
- Take large sets of Boolean CNFs (e.g. $(v_1 \vee v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$)

What's a SAT (or SMT) solver?

■ SAT solvers

- Only operate on Boolean logic
- Take large sets of Boolean CNFs (e.g. $(v_1 \vee v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$)
- Try to come up with inputs so the formula returns true (or fail and report that the formula is “unsatisfiable”, which means that it's impossible for the formula to be true)

What's a SAT (or SMT) solver?

- SAT solvers
 - Only operate on Boolean logic
 - Take large sets of Boolean CNFs (e.g. $(v_1 \vee v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$)
 - Try to come up with inputs so the formula returns true (or fail and report that the formula is “unsatisfiable”, which means that it's impossible for the formula to be true)
- SMT solvers

What's a SAT (or SMT) solver?

■ SAT solvers

- Only operate on Boolean logic
- Take large sets of Boolean CNFs (e.g. $(v_1 \vee v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$)
- Try to come up with inputs so the formula returns true (or fail and report that the formula is “unsatisfiable”, which means that it's impossible for the formula to be true)

■ SMT solvers

- “Satisfiability modulo theories” - a fancy term for a piece of software that uses a SAT solver to act as a theorem prover

What's a SAT (or SMT) solver?

■ SAT solvers

- Only operate on Boolean logic
- Take large sets of Boolean CNFs (e.g. $(v_1 \vee v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$)
- Try to come up with inputs so the formula returns true (or fail and report that the formula is “unsatisfiable”, which means that it's impossible for the formula to be true)

■ SMT solvers

- “Satisfiability modulo theories” - a fancy term for a piece of software that uses a SAT solver to act as a theorem prover
- Takes standard mathematical operations, bit operations, etc, and converts them to Boolean CNFs that a SAT solver can work with

What's a SAT (or SMT) solver?

■ SAT solvers

- Only operate on Boolean logic
- Take large sets of Boolean CNFs (e.g. $(v_1 \vee v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$)
- Try to come up with inputs so the formula returns true (or fail and report that the formula is “unsatisfiable”, which means that it's impossible for the formula to be true)

■ SMT solvers

- “Satisfiability modulo theories” - a fancy term for a piece of software that uses a SAT solver to act as a theorem prover
- Takes standard mathematical operations, bit operations, etc, and converts them to Boolean CNFs that a SAT solver can work with
- Again, tries to come up with inputs so that the formula is satisfiable

What's a SAT (or SMT) solver?

- SAT solvers
 - Only operate on Boolean logic
 - Take large sets of Boolean CNFs (e.g. $(v_1 \vee v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$)
 - Try to come up with inputs so the formula returns true (or fail and report that the formula is “unsatisfiable”, which means that it’s impossible for the formula to be true)
- SMT solvers
 - “Satisfiability modulo theories” - a fancy term for a piece of software that uses a SAT solver to act as a theorem prover
 - Takes standard mathematical operations, bit operations, etc, and converts them to Boolean CNFs that a SAT solver can work with
 - Again, tries to come up with inputs so that the formula is satisfiable
- Symbolic execution engines

What's a SAT (or SMT) solver?

■ SAT solvers

- Only operate on Boolean logic
- Take large sets of Boolean CNFs (e.g. $(v_1 \vee v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$)
- Try to come up with inputs so the formula returns true (or fail and report that the formula is “unsatisfiable”, which means that it's impossible for the formula to be true)

■ SMT solvers

- “Satisfiability modulo theories” - a fancy term for a piece of software that uses a SAT solver to act as a theorem prover
- Takes standard mathematical operations, bit operations, etc, and converts them to Boolean CNFs that a SAT solver can work with
- Again, tries to come up with inputs so that the formula is satisfiable

■ Symbolic execution engines

- Convert CPU instructions into a SMT solver language

In case this all seemed too abstract...

I get by with a little help from my friends in *NP*

SMT solvers let you write a type of SQL that can be used to solve certain
hard* problems

In case this all seemed too abstract...

I get by with a little help from my friends in *NP*

SMT solvers let you write a type of SQL that can be used to solve certain
hard* problems

* hard, meaning “NP-complete”

In case this all seemed too abstract...

I get by with a little help from my friends in *NP*

SMT solvers let you write a type of SQL that can be used to solve certain hard* problems

* hard, meaning “NP-complete”

The SMT query itself can be constructed from a language like Python, C, or even raw x86 assembly

In case this all seemed too abstract...

I get by with a little help from my friends in *NP*

SMT solvers let you write a type of SQL that can be used to solve certain
hard* problems

* hard, meaning “NP-complete”

The SMT query itself can be constructed from a language like Python, C,
or even raw x86 assembly

If there's one thing you should take from this presentation: SMT solvers
are tools for solving constraint problems, just like SQL is a tool for
making sense of large amounts of data

■ Z3 (SMT)

- Open source, developed by Microsoft
- Has its own SQL-like language, with Python bindings
- Contains its own SAT solver

Modern SMT solvers and symbolic execution engines

- Z3 (SMT)
 - Open source, developed by Microsoft
 - Has its own SQL-like language, with Python bindings
 - Contains its own SAT solver
- KLEE (symbolic execution)
 - Used to instrument code that can be recompiled to LLVM IR
 - C bindings
 - Z3, Kleaver, or SMTLib used as a backend

Modern SMT solvers and symbolic execution engines

- Z3 (SMT)
 - Open source, developed by Microsoft
 - Has its own SQL-like language, with Python bindings
 - Contains its own SAT solver
- KLEE (symbolic execution)
 - Used to instrument code that can be recompiled to LLVM IR
 - C bindings
 - Z3, Kleaver, or SMTLib used as a backend
- Ponce (symbolic execution)
 - An IDA Pro plugin for x86 and x86_64 instruction sets
 - Z3 used as a backend

Modern SMT solvers and symbolic execution engines

- **Z3** (SMT)
 - Open source, developed by Microsoft
 - Has its own SQL-like language, with Python bindings
 - Contains its own SAT solver
- KLEE (symbolic execution)
 - Used to instrument code that can be recompiled to LLVM IR
 - C bindings
 - Z3, Kleaver, or SMTLib used as a backend
- Ponce (symbolic execution)
 - An IDA Pro plugin for x86 and x86_64 instruction sets
 - Z3 used as a backend

We'll be mainly focusing on using Z3, since all the other tools use it under the hood (and it's awesome)

Back to our sudoku...

How do you create a hexadoku puzzle out of thin air with Z3?

Back to our sudoku...

How do you create a hexadoku puzzle out of thin air with Z3?

Creating the board

```
import z3

# Define a 16x16 Sudoku
order = 4; side = order ** 2

# Create a solver and a 16x16 board (list of lists)
smt = z3.Solver()
cells = [list() for row in range(side)]
for r in range(side):
    for c in range(side):
        cells[r].append(z3.Int('cell_%d_%d' % (r, c)))
```

Telling z3 how to play sudoku

We'll define our first rule: each cell has to be between 0 and 15

Telling z3 how to play sudoku

We'll define our first rule: each cell has to be between 0 and 15

The most basic of Sudoku rules

```
for r in range(side):
    for c in range(side):
        # Each cell must be between 0 and 15.
        # Also, this is our first constraint!
        # Add it to the solver with `smt.add`.
        smt.add(z3.And(cells[r][c] >= 0, cells[r][c] < side))
```

Telling z3 how to play sudoku

Now, the rules everyone knows

Telling z3 how to play sudoku

Now, the rules everyone knows

Rows and columns

```
# Rows must have unique values
for r in range(side):
    this_row = [cells[r][c] for c in range(side)]

    # Ensures that all values are represented only once
    smt.add(z3.Distinct(*this_row))

# Columns must have unique values
for c in range(side):
    this_col = [cells[r][c] for r in range(side)]
    smt.add(z3.Distinct(*this_col))
```


Telling z3 how to play sudoku

Mini-boxes are slightly more complicated

Telling z3 how to play sudoku

Mini-boxes are slightly more complicated

Mini-boxes

```
# Each mini-box must have unique values
for r in range(0, side, order):
    for c in range(0, side, order):
        selected = []
        for i in range(box_size):
            for j in range(box_size):
                selected.append(cells[r + i][c + j])

        smt.add(z3.Distinct(*selected))
```

Telling z3 how to play *our version of* sudoku

Now for the initial values

... dc858 isn't sudoku-able, but dc619 is

Telling z3 how to play *our version of* sudoku

Now for the initial values

... dc858 isn't sudoku-able, but dc619 is

Initial values

```
# Now, define the initial values
```

```
smt.add(cells[0][0] == 0xd)
```

```
smt.add(cells[0][1] == 0xc)
```

```
smt.add(cells[0][2] == 0x6)
```

```
smt.add(cells[0][3] == 0x1)
```

```
smt.add(cells[0][4] == 0x9)
```

```
# ... etc.
```

Telling z3 to start solving

This will take a little bit of time

Telling z3 to start solving

This will take a little bit of time

Start solving!

```
if smt.check() == z3.unsat:
    print('Sudoku is impossible to solve :(')
else:
    # Retrieve each cell from the model as a python long
    model = smt.model()
    result = [list() for row in range(side)]
    for r in range(side):
        for c in range(side):
            cell = model[r][c]
            print('%x ' % model[cell].as_long(), end='')
        print()
```

The results...

d	c	6	1	9	0	e	b	5	7	3	2	4	8	f	a
5	8	a	4	f	c	2	d	9	0	b	e	3	7	6	1
2	b	9	7	5	3	6	1	4	8	a	f	d	0	e	c
0	3	f	e	4	a	8	7	d	c	6	1	9	b	2	5
8	d	c	6	1	9	3	2	b	5	e	7	0	4	a	f
b	1	5	0	d	8	c	6	f	a	9	4	7	2	3	e
9	a	4	f	7	5	0	e	2	3	1	8	b	d	c	6
7	2	e	3	b	4	a	f	0	d	c	6	1	9	5	8
e	4	d	c	6	1	9	0	a	b	2	5	8	f	7	3
1	9	8	2	3	7	5	4	e	6	f	0	a	c	d	b
f	6	7	a	2	d	b	c	1	9	8	3	5	e	0	4
3	5	0	b	a	e	f	8	7	4	d	c	6	1	9	2
a	e	3	d	c	6	1	9	8	f	7	b	2	5	4	0
6	0	2	5	e	b	d	3	c	1	4	9	f	a	8	7
c	7	1	9	8	f	4	5	6	2	0	a	e	3	b	d
4	f	b	8	0	2	7	a	3	e	5	d	c	6	1	9

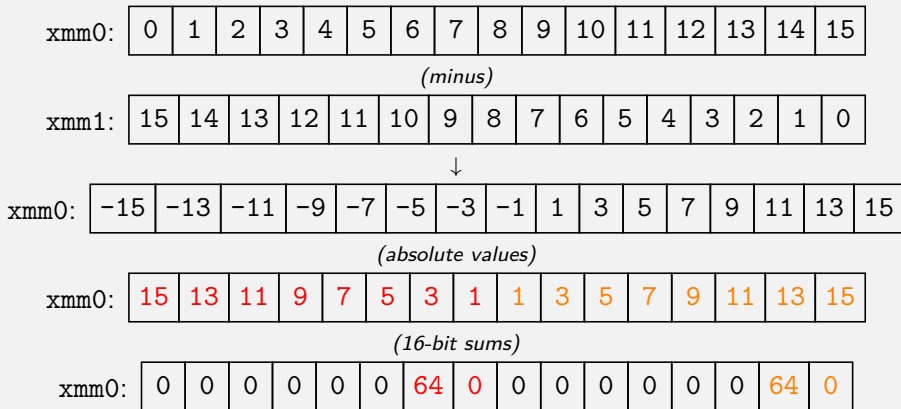
The power of SMT solvers

Maybe familiar to anyone who's tried Prolog

We only had to **define the constraints of the puzzle we wanted**, and Z3 found one satisfying those constraints

Constraint solving our boot sector

“Hey, Z3, I want you to find me input values that, after being put through this crazy operation (among other things), end up being equal to the ones stored in the boot sector”



Useful primitives

- We need to implement the pshufd and psadbw x86 instructions in Z3
- Documentation at <http://x86.renejeschke.de> already provides us pseudocode for these instructions

Operation
Destination[0..31] = (Source >> (Order[0..1] * 32))[0..31];
Destination[32..63] = (Source >> (Order[2..3] * 32))[0..31];
Destination[64..95] = (Source >> (Order[4..5] * 32))[0..31];
Destination[96..127] = (Source >> (Order[6..7] * 32))[0..31];

Useful constraints

The flag looks like `flag{some_text_here}`, but the word “flag” is chopped off before it's loaded into an XMM register, so the first and last characters of the remaining 16 are { and }, respectively.

Useful constraints

The flag looks like `flag{some_text_here}`, but the word “flag” is chopped off before it's loaded into an XMM register, so the first and last characters of the remaining 16 are { and }, respectively. This may not look like much, but we already have a few constraints...

Initial constraints

```
import z3
smt = z3.Solver()
flag = [z3.BitVec('flag_%d' % i, 8) for i in range(16)]

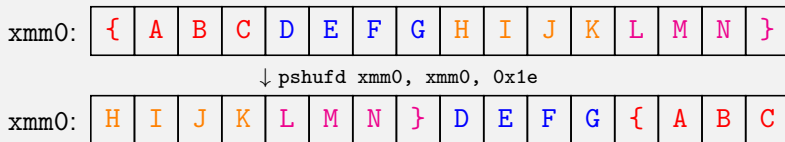
# Flag must start with '{' and end with '}', and the rest must be ASCII
smt.add(z3.And(flag[0] == ord('{'), flag[-1] == ord('}')))
for i in range(1, 15):
    smt.add(z3.And(flag[i] >= 32, flag[i] < 127))
```

Understanding the hash function's word shuffling

The flag is initially shuffled using the pshufd instruction; create a new constraint (“shuf”) that just describes the permutation

Understanding the hash function's word shuffling

The flag is initially shuffled using the pshufd instruction; create a new constraint ("shuf") that just describes the permutation



Implementing the hash function's word shuffling

Implementing the word shuffling (pshufd)

```
# Create the permuted flag, add trivially true constraints
shuf = [z3.BitVec('pshufd_%d' % i, 8) for i in range(16)]

smt.add(
    z3.And(shuf[0] == flag[8], shuf[1] == flag[9],
           shuf[2] == flag[10], shuf[3] == flag[11], ...)
)
```

Implementing the hash function's bitmasking

For each of the hash function's 8 rounds, a different mask is applied to the shuffled value, so create another set of intermediate values called “masked”

Implementing the hash function's bitmasking

For each of the hash function's 8 rounds, a different mask is applied to the shuffled value, so create another set of intermediate values called "masked"

Implementing the masking operation (andps)

```
# Masks for each round:
# ffffffff ffffffff00 ffffffff ffffffff00
# ffffffff ffff00ff ffffffff ffff00ff
# ffffffff ff00ffff ffffffff ff00ffff
# ffffffff 00ffffff ffffffff 00ffffff
# ... etc.
mask = [0x00 if i == round else 0xff for i in range(8)] * 2
masked = [z3.BitVec('masked_%d_%d' % (round, i), 8) for i in range(16)]
for i in range(16):
    smt.add(masked[i] == (shuf[i] & mask[i]))
```

Implementing the packed sum of absolute differences

A single x86 instruction turned into like 30 lines of Python...

Implementing the packed sum of absolute differences

A single x86 instruction turned into like 30 lines of Python...

Implementing psadbw (greatly simplified; trust me, it's nasty)

```
ps_lo, ps_hi = z3.BitVec('psadbw_lo', 16), z3.BitVec('psadbw_hi', 16)
```

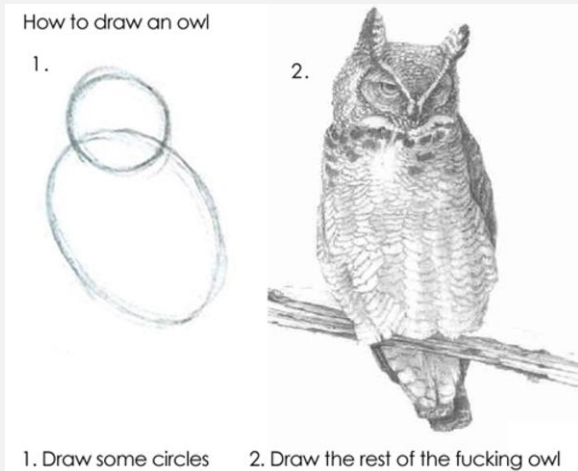
```
smt.add(ps_lo == (  
    Abs(iv[0] - masked[0]) + Abs(iv[1] - masked[1]) +  
    Abs(iv[2] - masked[2]) + Abs(iv[3] - masked[3]) + ...  
)) # and repeat for ps_hi. Assumes that we have an `Abs` helper.
```

```
res = [z3.BitVec('psadbw_%d' % i, 8) for i in range(16)]  
smt.add(res[0] == ps_lo & 0xff)  
smt.add(res[1] == (ps_lo >> 8) & 0xff)  
smt.add(res[2] == 0) # ... etc. Read Intel docs if you care.
```

And the rest of the algorithm...

AKA: continuing to give up on explaining the details

- Would be boring to present, since there were a few arbitrary things thrown into the challenge that made it deliberately more annoying to reverse-engineer
- ... Like x86 tricks where they loaded a register like `eax` and used `ah` and `al` backwards
- Not to mention that the algorithm has to run for 8 iterations...



Once we've got the constraints defined...

Now, we can solve it as usual, and retrieve the flag from the model

Once we've got the constraints defined...

Now, we can solve it as usual, and retrieve the flag from the model

Solving for the flag

```
if smt.check() == z3.unsat:
    print('Formula was unsatisfiable :-(')
else:
    print('Formula was satisfiable.')
    model = smt.model()
    flag_str = 'flag'
    for i in range(16):
        flag_str += chr(model[flag[i]].as_long())

    print(flag_str)
```

Run it, and...

```
Formula was satisfiable.  
flag{ D`?T\B?3F> P` }
```

Does it work?

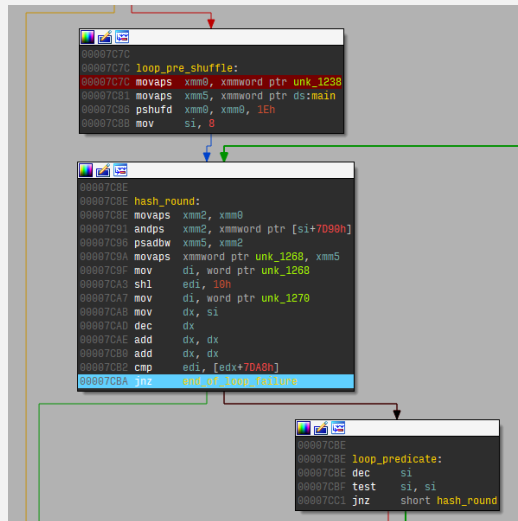
Well, no

```
== ENTER FLAG ==  
flag{ D"?T\B?3F>~P`}
```

```
!! WRONG FLAG !!  
flag{ D"?T\B?3F>~P`}
```


Does it work?

But, using IDA's debugger and setting breakpoints, we see that it got through the first round of the hash function, but failed on the second - turns out, we forgot to solve the remaining rounds



- Baby steps - try to get a satisfiable formula first, then focus on making it the *correct* satisfiable formula

Debugging SMT solvers

- Baby steps - try to get a satisfiable formula first, then focus on making it the *correct* satisfiable formula
- If you're getting unsatisfiable, make sure you're not overconstraining (i.e. assuming that the flag only contains letters)

Debugging SMT solvers

- Baby steps - try to get a satisfiable formula first, then focus on making it the *correct* satisfiable formula
- If you're getting unsatisfiable, make sure you're not overconstraining (i.e. assuming that the flag only contains letters)
- If you're reimplementing some piece of code in Z3, make sure its flow matches the original's

Debugging SMT solvers

- Baby steps - try to get a satisfiable formula first, then focus on making it the *correct* satisfiable formula
- If you're getting unsatisfiable, make sure you're not overconstraining (i.e. assuming that the flag only contains letters)
- If you're reimplementing some piece of code in Z3, make sure its flow matches the original's
 - Match control structures as faithfully as possible

Debugging SMT solvers

- Baby steps - try to get a satisfiable formula first, then focus on making it the *correct* satisfiable formula
- If you're getting unsatisfiable, make sure you're not overconstraining (i.e. assuming that the flag only contains letters)
- If you're reimplementing some piece of code in Z3, make sure its flow matches the original's
 - Match control structures as faithfully as possible
 - Use Python functions to implement complex behaviors like Intel SSE2 instructions with the correct series of `smt.add` calls

```
Formula was satisfiable.  
flag{4e@alz_p%Z/TnW}
```

```
Formula was satisfiable.  
flag{4e@alz_p%Z/TnW}
```

And, it got through 6 out of the 8 rounds this time!


```
Formula was satisfiable.  
flag{4e@alz_p%Z/TnW}
```

And, it got through 6 out of the 8 rounds this time!
... But, it returned unsatisfiable when trying to solve for 7 or 8

Typos are your worst enemy

Spot the difference...

What I typed

```
outputs = (  
    0x02df028f,  
    0x0290025d,  
    0x02090221,  
    0x027b0279,  
    0x01f90233,  
    0x025e0291,  
    0x02290255,  
    0x02110270  
)
```

What I meant

```
outputs = (  
    0x02df028f,  
    0x0290025d,  
    0x02090221,  
    0x027b0278, # single bit typo  
    0x01f90233,  
    0x025e0291,  
    0x02290255,  
    0x02110270  
)
```

```
Formula was satisfiable.  
flag{4r3alz_m0d3_y0}
```

After fixing the typo...

Formula was satisfiable.

flag{4r3alz_m0d3_y0}

»»»» CORRECT! ««««

flag{4r3alz_m0d3_y0}

So, what did we learn?

- Emulators like QEMU can be used to help debug code for non-native architectures

So, what did we learn?

- Emulators like QEMU can be used to help debug code for non-native architectures
- Intel x86 vector instructions are complex, but we can get past the complexity by following the documentation

So, what did we learn?

- Emulators like QEMU can be used to help debug code for non-native architectures
- Intel x86 vector instructions are complex, but we can get past the complexity by following the documentation
- SMT solvers can be used to query constraints, similarly to how you can use SQL to query lots of data

The end!

- This presentation is built on the shoulders of giants; check out Dennis Yurichev's fantastic writeup on SAT and SMT solvers
 - https://yurichev.com/writings/SAT_SMT_draft-EN.pdf
- Z3 code for generating sudokus is at <https://goo.gl/ZC1546>
- Z3 code for solving the CTF challenge is available at <https://goo.gl/cJPJDN>
- Questions?