Curiouser and curiouser

A selection of malware obfuscation techniques

Cindy Eisner
Senior Technical Staff Member
IBM Research - Haifa
December 2016

© 2016 IBM Corporation

IBM

Obfuscate

ob·fus·cate

/ˈäbfəˌskāt/

render obscure, unclear, or unintelligible

render obscure, unclear, or unintelligible.

"the spelling changes will deform some familiar words and obfuscate their etymological origins"

synonyms: obscure, confuse, make unclear, blur, muddle, complicate, overcomplicate, muddy, cloud, befog

"mere rationalizations to obfuscate rather than clarify the real issue"

antonyms: clarify

- bewilder (someone).

"it is more likely to obfuscate people than enlighten them"

synonyms: bewilder, mystify, puzzle, perplex, confuse, baffle, confound, bemuse, befuddle, nonplus; *informal* flummox

"her work became more and more obfuscated by mathematics and jargon"

Why obfuscate?

→ To make code hard to reverse engineer

Origin

late Middle English: from late Latin *obfuscat-* 'darkened,' from the verb *obfuscare*, based on Latin *fuscus* 'dark.'

IBM

Stuff that malware does

- Taking advantage of vulnerabilities to infect
- Unpacking
- Anti-debugging, anti-research
- **Obfuscation**
- Malicious payload

About the examples

- **Always**, we have **machine** code
- However, to **clarify**, I've **lifted** some of them to **c**
  - And **simplified** considerably

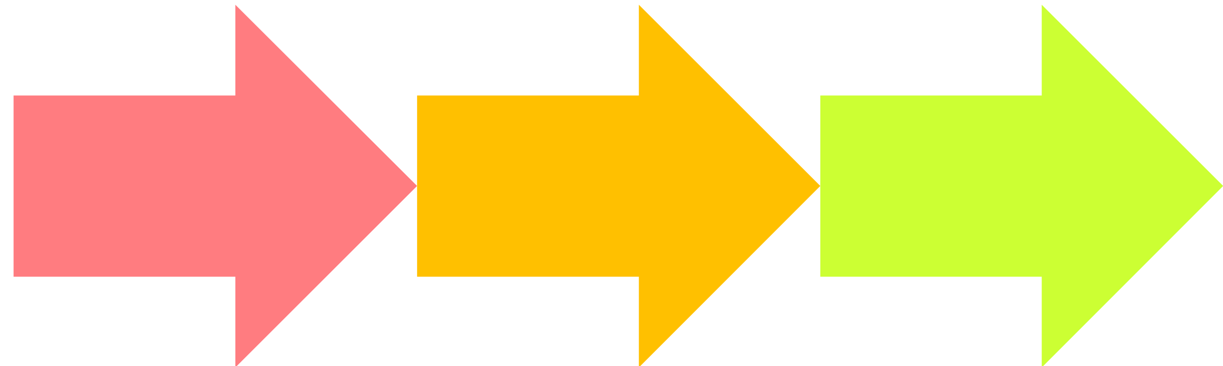
Example program

```
if (beingAnalyzed())
        fatal_error("this program is being analyzed");
else
        malicious();
```

IBM

Shift right

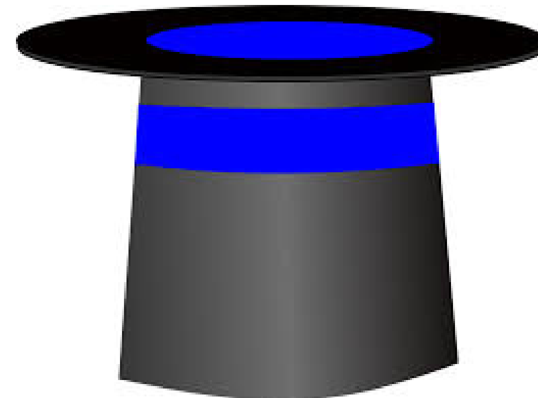
```
x = beingAnalyzed();
wasteTime(); /* but make it look like work */
if (x != 0)
        fatal_error("this program is being analyzed");
else
        malicious();
```



IBM

“Look Ma, no hands!”

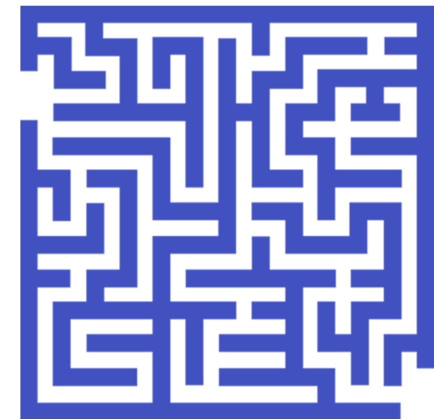
```
hdc = GetDC(makeAnInvisibleWindow());
SetBkColor(hdc,beingAnalyzed());
wasteTime(); /* but make it look like work */
if (SetBkColor(hdc, RGB(6,7,8)) != 0)
        fatal_error("this program is being analyzed");
else
        malicious();
```



IBM

And why make it easy?

```
hdc = GetDC(makeAnInvisibleWindow());
SetBkColor(hdc,beingAnalyzed());
wasteTime(); /* but make it look like work */
if (SetBkColor(hdc, RGB(6,7,8)) != 0)
      while(1) {wasteTime();}
else
      malicious();
```



Infinite loops

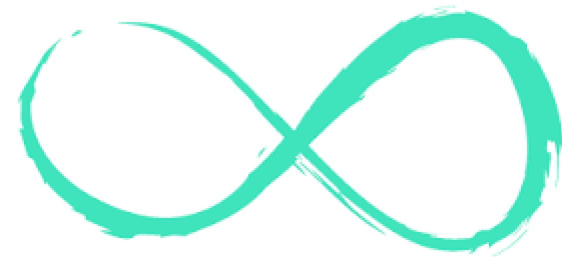
Thread 1:

```
a: inc eax
   jmp a
b: more code…
c: inc eax
   jmp d
d: more code…
e: inc eax
   jmp e
f:  more code…
```

Thread 97:

Overwrite single byte ‘c’ with ‘d’

etc. etc.



IBM

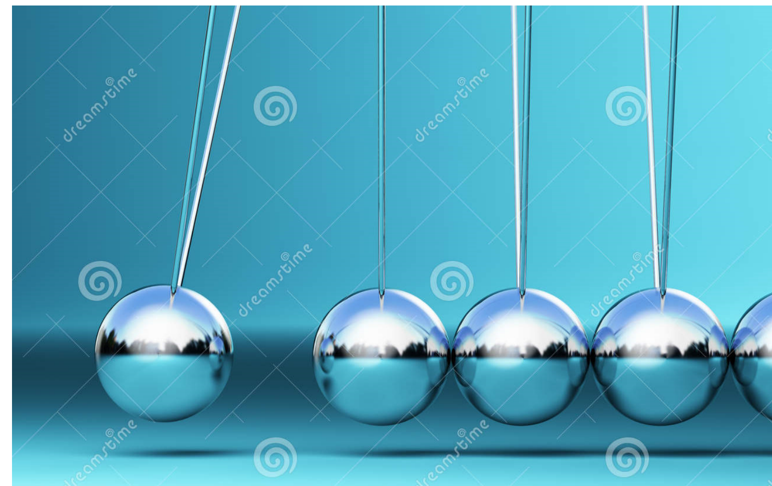
A way to waste time (but looks like work)

```
for (i=0; i<100000000; i++)
  {y = copyit(x); z = copyit(y); x = copyit(z);}

int copyit(int i) {
  static int j=0, k=0;
  if (j>0) {j = i – 45; k = j + 10;}
  else if (k%7) {j = i – 100; k = j + 65;}
  else {j = i * 3; k = (j/3) – 35;}
  return (k + 35);
}
```

Another way to waste time

- Spawn lots of threads
- Pass events back and forth



IBM

Camouflaging API calls

```
Kernel32.GetModuleHandleA
7dd71245    mov   edi, edi
7dd71247    push  ebp
7dd71248    mov   ebp, esp
7dd7124a    pop   ebp
7dd7124b    jmp   short GetModuleHandleA_0
```

- So, call **7dd7124b** directly (anonymous function)

Flouting coding conventions

```
push eax
ret
```

```
     push eax (=a)
     push ebx (=b)
     push ecx (=c)
     ret
     …
c:   ret
     …
b:   ret
     …
a:   …
```

```
push eax
jmp <somesystemcall>
```

```
     cmp eax, ecx
     je a
     jmp b
     …
b:   jge a
     jmp c
     …
c:   jmp a
```

```
     call p
.
.
.
p:   pop eax
```

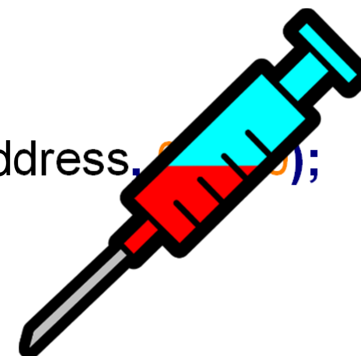
Not in my backyard

```
CreateProcess
("c:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe",
NULL,NULL,NULL,FALSE,0,NULL,NULL,&sinfo,&pinfo);

BYTE *startaddress = (BYTE *)
VirtualAllocEx
(pinfo.hProcess, 0, size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

WriteProcessMemory
(pinfo.hProcess, startaddress, &localbuf, size, &byteswritten);

CreateRemoteThread
(pinfo.hProcess, 0, 0, (LPTHREAD_START_ROUTINE)startaddress, 0 [illegible] 0);
```



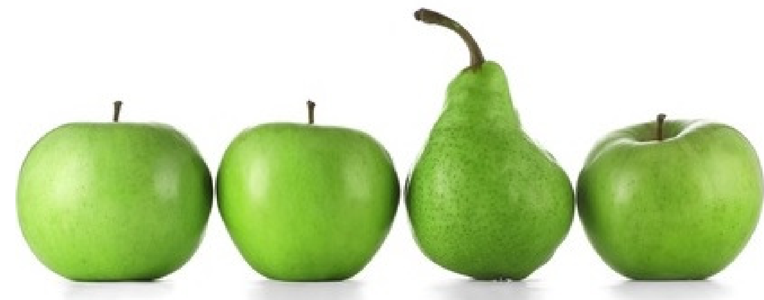
Where's Wally?

```
ret = NtQuerySystemInformation(x, y, z, w);
```

```
typedef int (WINAPI*ftype)(PVOID a, PVOID b, PVOID c, PVOID d);
HINSTANCE lib = LoadLibrary(TEXT("ntdll.dll"));
ftype fp = (ftype) GetProcAddress(lib, "NtQuerySystemInformation");
ret = (fp) (x, y, z, w);
```

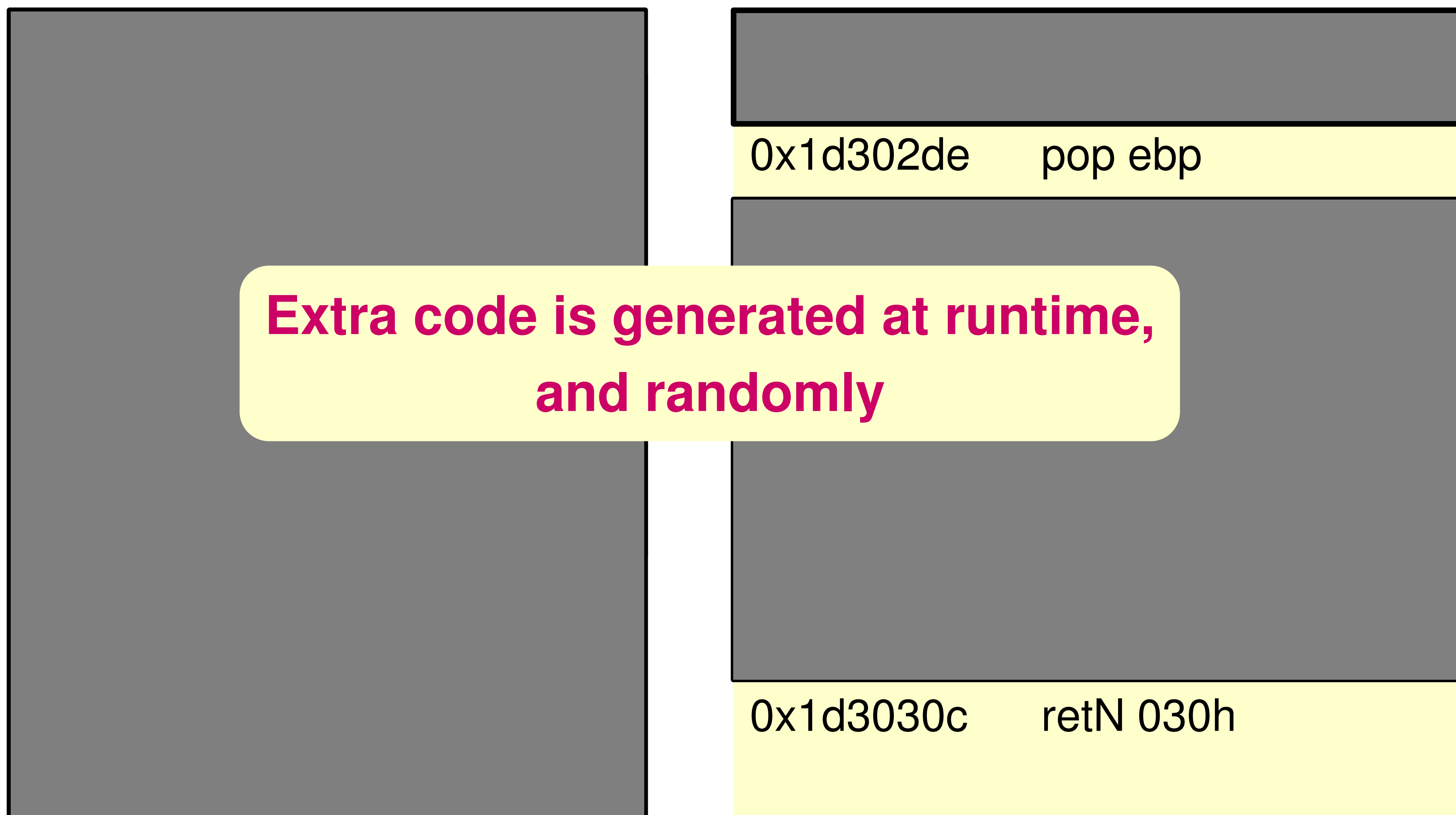
Does anybody see the malicious code?

```
try {
  string s1 = "1.052033", s2 = "e+00", s3 = "3";
  for (int i = 0; i<1000; i++) {
    for (int j = 0; j<1000;j++) {
      long double f = stof(s1+s2+s3); s1 += s3;
    }
    int t = stoi(s3); t *= 2; s3 = to_string(t);
  }
}
catch (...) { f(); }
```

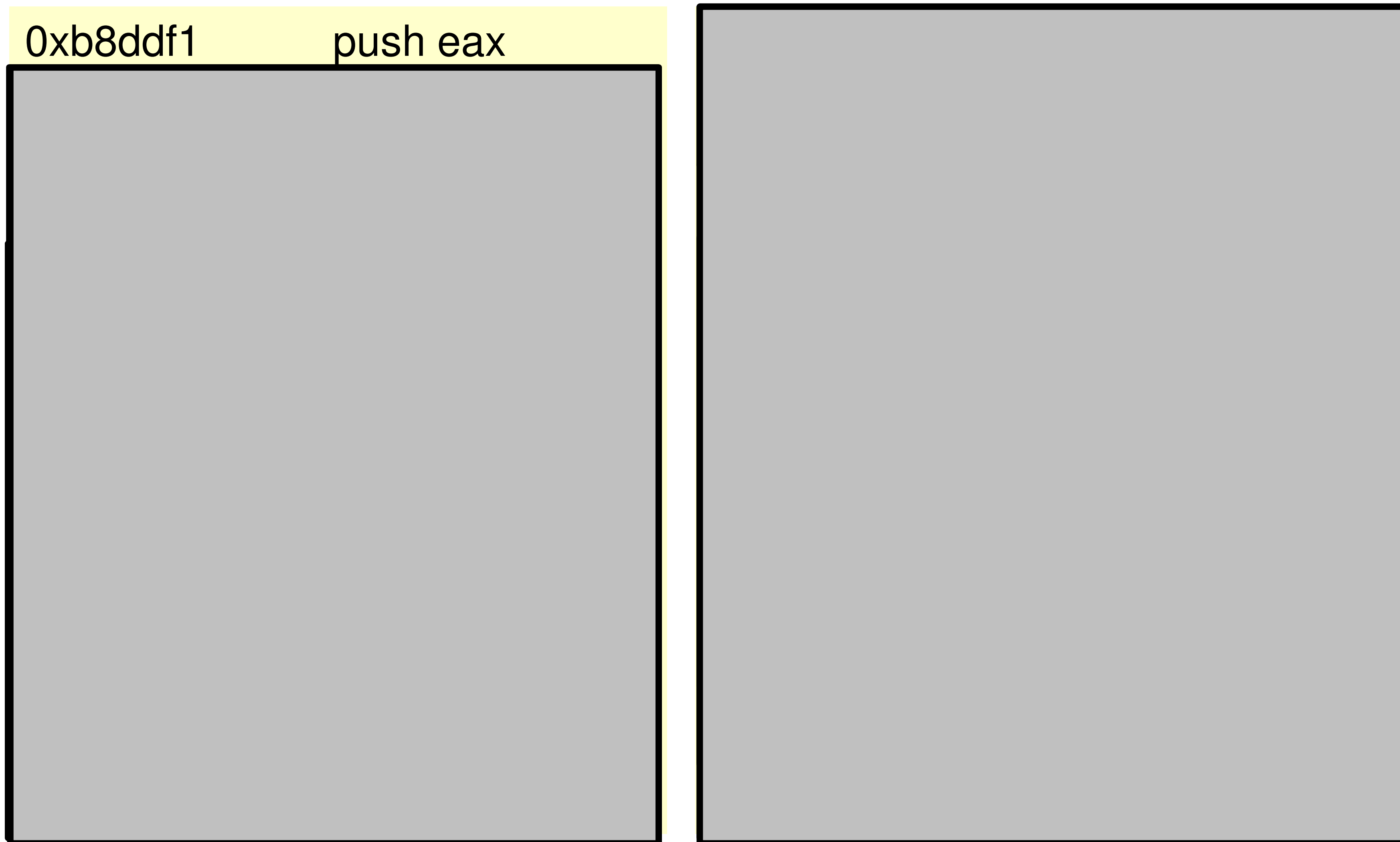


IBM

Wasting time and camouflaging ops



Time stamp confusion



Thank You

Backup slides

IBM

Packing

Packing/unpacking

- **Pack**
  - To compress an executable file
- **Unpack**
  - To run a second executable that:
    - Decompresses the compressed file
    - Runs it

IBM

Packing is not always malicious

- Reduce needed **storage** or download **time**
- Protect **intellectual property** through obfuscation
  - Because packed code is harder to reverse engineer
  - Especially if combined with encryption

But often it is malicious

- Protect **intellectual property** through obfuscation
  - Because packed code is harder to reverse engineer
  - Especially if combined with encryption
- Render **signatures** useless
  - Because once my malware is identified, all I have to do is repack

And not always easy to understand

Example (based on Shylock/Caphaw)

- Take a piece of malware (call it **A**), encrypt it (giving **B**)
- Then, write a program (**C**) that generates the encrypted code
- Encrypt **C**, write it into your data section
- Write a program (**D**) that:
  - Allocates space in memory
  - Decrypts its data section there (this will give you **C**)
  - Allocates more space in memory, in which it writes code (**E**) that:
    - Erases (**D**)
    - Runs **C** to generate **B** (overwriting the zeroed-out **D**)
    - Allocates more memory
    - Decrypts **B** into the newly alloced memory (this will give you **A**)
    - Runs the result (**A** – the original malicious code)
  - Runs **E**

Anti-debugging

IBM

Anti-debugging, anti-research

- Straightforward packed code is **fairly easy** to unpack
  - Just watch what it does, e.g., in a **debugger** or in an **emulator**
  - Watch carefully! (i.e., in a **VM**)
- **Anti-debugging** techniques are intended to make unpacking more **difficult**
  - Exploit **subtle differences** between environments
    - Under debugger **vs.** independent run
    - In virtual **vs.** real machine
  - Or just make it **confusing**, e.g.
    - Lots of **jumps**
    - Lots of **threads**

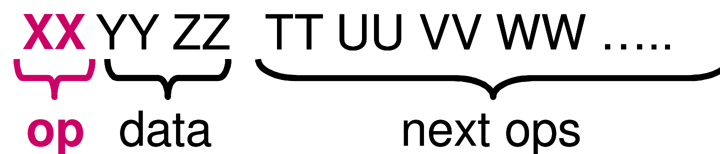
IBM

Exceptions

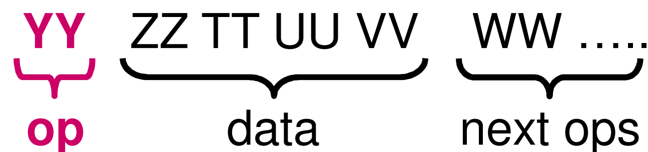
- Register an exception handler, then generate an exception **on purpose**
  - **Debuggers** are given first chance to handle exception
  - So behavior is **different** under debugger vs. without debugger (there are **workarounds** – the point is just to make it hard for the reverser)
- Exception handler has access to all registers, including **EIP**
  - So has full control over **where** it returns and in **what** state
  - Lets you do **funky** stuff

Funky things you can do with exceptions

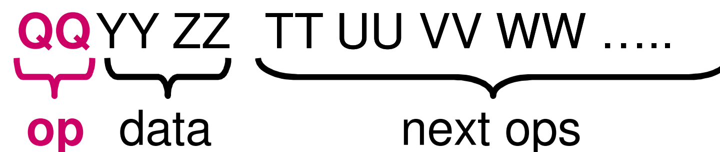
- Suppose **EIP** points here: XX YY ZZ TT UU VV WW …..
- And suppose operation beginning with **XX** consumes 3 bytes total, like this:



- Furthermore, assume operation has been written to generate an **exception**
- In the exception handler, **increment** EIP, so now the operation is this:



- Or you can just **overwrite** the opcode



IBM

Funky things, continued

- **Change** the value of any general purpose **register**
- Or any **flag**
- Makes it hard to **read** the code, hard to debug, and hard for **static disassemblers**
- You can of course **nest** exceptions…

IBM

Checking for debugger, VM

- **IsDebuggerPresent(),** or access **bit** accessed by IsDebuggerPresent() (bit 3 of structure pointed to by FS:[30h] (**Thread Information Block**))
- **NtQueryInformationProcess()**, with certain parameters reveals **debug port**
- Check for presence of various **files**, **windows**, **registry keys**, e.g.
  - File "**\\.\SICE**": indicates SoftICE kernel debugger
  - Window of class "**ollydbg**" indicates OllyDbg binary debugger
  - Registry key named "**Software\Wine**" indicates Wine (un)emulator
- Check for environment variables, e.g.
  - Presence of **WLNumDLLsProt** indicates to malware it is being watched
- Opcode 0f 00 (**SLDT**) returns 0 on Windows, non-zero on VMWARE
- Calculate **md5** of code, compare to expected (finds software breakpoints)
- There are **MANY** other ways

Timing

- There are various ways to check the **time**
- Can be used to **distinguish** run under debugger or VM from independent run
- **Phantom** checks the time frequently, but then **discards** result
  - Seems to be a way to **confuse** the human, who might waste time faking time