An Introduction to Native Postmortem Debugging

"*I think it's dead, Jim*"

Dr. L McCoy, Star date 3196

"*Hex codes never lie*"

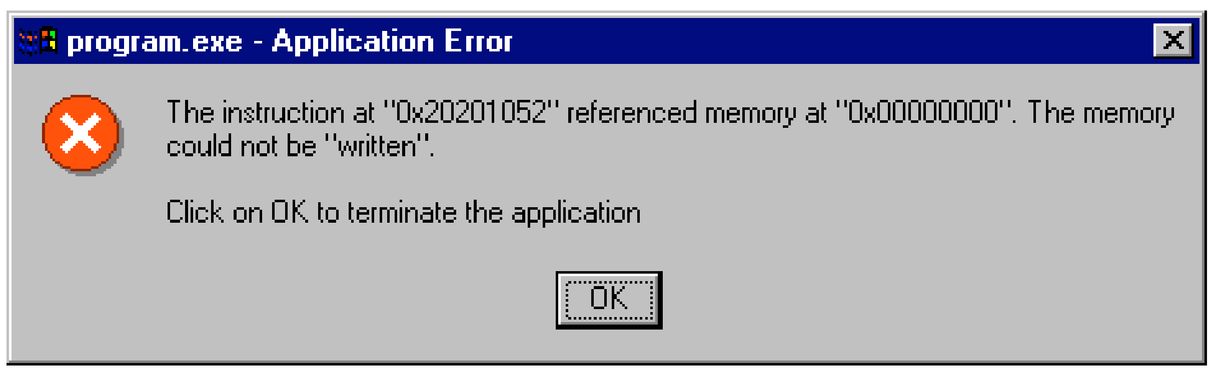
Mark Bartosik, 2003 AD

This talk contains intel x86 and Microsoft OS and compiler specific details.
Helps if you know a little assembler (understand "**push EAX**")

Motivation ?

- Software phases
  - Requirement
  - Design
  - Code
  - Test
  - **Debug**
  - **Support & maintenance**

Debug This!



Call stack:

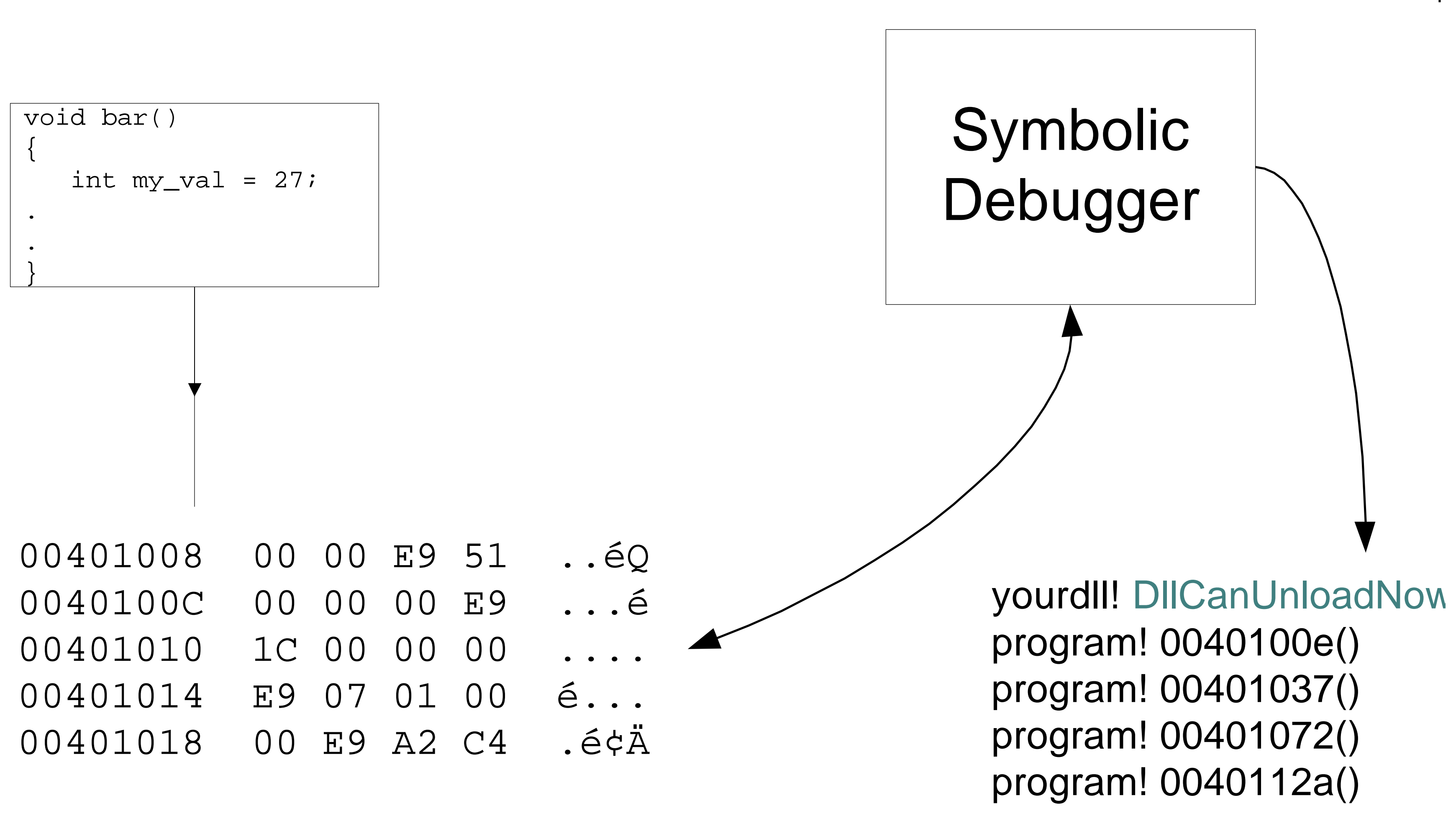
```
Somedll! Dllcanunloadnow()
Program! 0040100e()
Program! 00401037()
Program! 00401072()
Program! 0040112a()
```

Importance of a Stack Trace

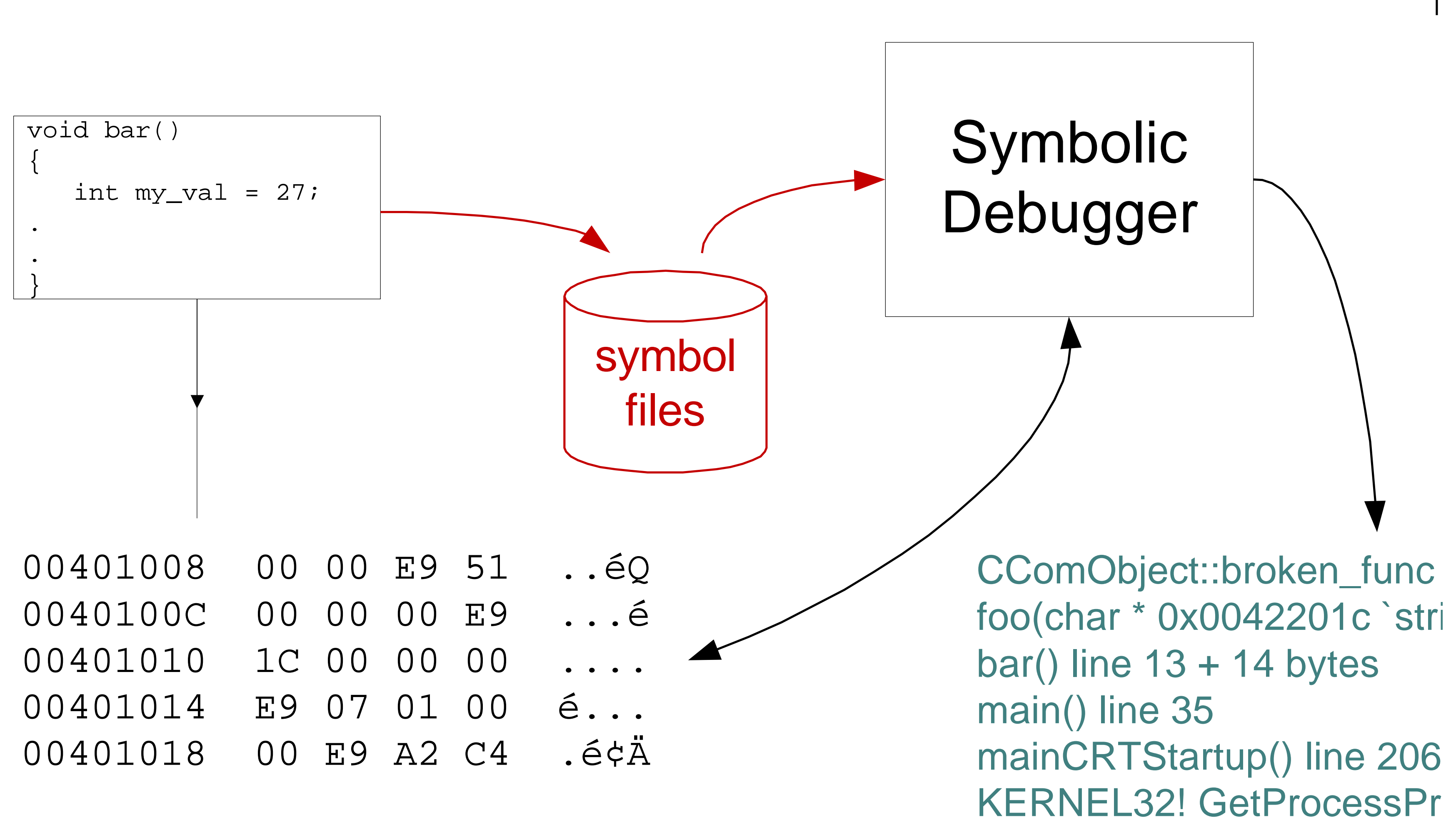
Information on the stack:

- History of which functions executed
  what the program / thread was doing before it crashed
- Contents of automatic variables

Debugging without symbols

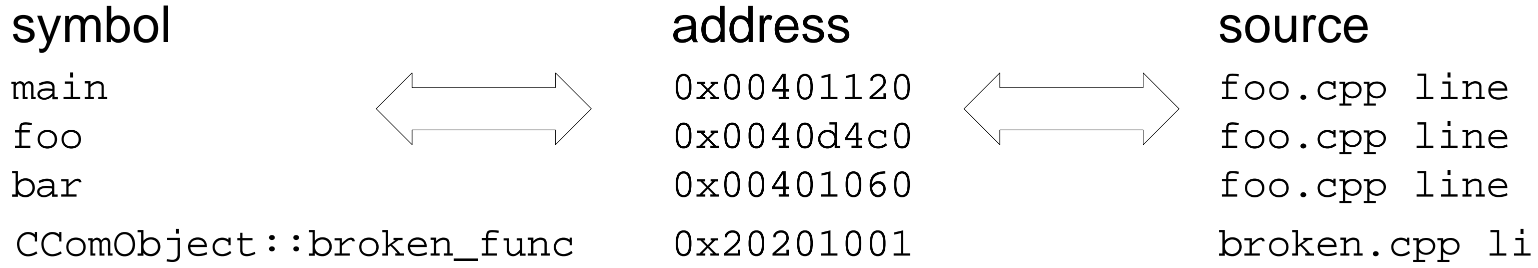


Debugging with symbols



Symbol file contents

- Addresses of functions, source code locations
- Stack decoding information
- Locations of variables
- Type information



Sources of symbols

- Symbol files -- the ideal source!
  (Generated by the compiler and linker -- **keep them safe**)

- Public exports __declspec(dllexport) a poor fallback
  C:\>dumpbin /exports your.dll

- Type libraries (only one debugger that I know uses these)
  See *Improve Your Debugging by Generating Symbols from COM Type Libraries,* Matt Pietrek MSJ March 1999

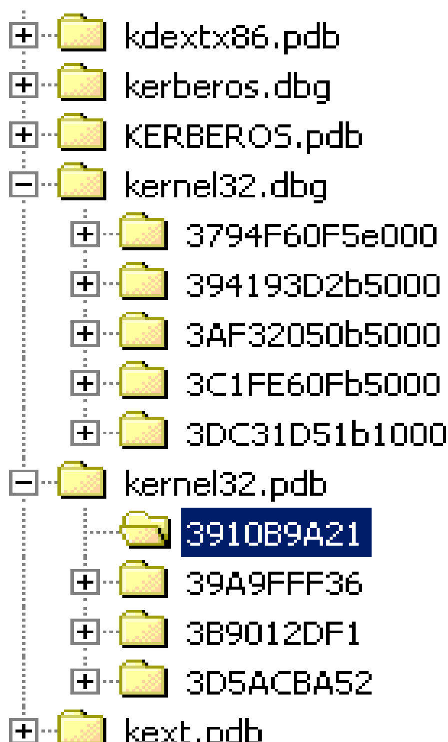
- .map files
  (limited human readable symbol files)

Symbol servers

How many versions of kernel32.dll are there out there?

How many versions of your.dll have you released?
How should you organize your symbol files?
Which symbol files should be on your debugger's symbol path?

A symbol server automatically provides the right symbols for the executable image.
Microsoft provides one accessible via HTTP.

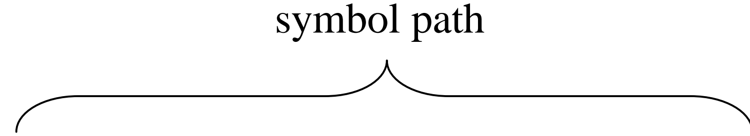


Tip: Take a look at symchk.exe, symstore.exe, ships with windbg
http://www.microsoft.com/ddk/debugging
http://www.microsoft.com/ddk/debugging/symbols.asp

Tip: Add this to your debugger's symbol path:
SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols

Symbol server support

- Visual Studio 5.0, 6.0 loads symbols only on startup
- Visual Studio 7.0 / .NET can load on demand, easy to configure
- Windbg 6.x has symbol server support
- Visual Studio 7.1



symchk /ie notepad.exe /s SRV*c:\temp*\\server\symbols

```
SRV*\\hpgs088fs1\symbols\microsoft\fixed_cache;
SRV*\\hpgs088fs1\symbols\microsoft\web_cache*http://msdl.microsoft.com/download/symbols
```

User / mini dump files

aka "core files"

A photograph of a process or part of a process

Microsoft: see MSDN MiniDumpWriteDump, and/or userdump, mscordmp

cygwin: see dumper, and error_start environment variable

- Stacks
- Stacks + data sections (global variables)
- All accessible memory
- Operating system handles

Can be loaded into a debugger, but you need the module files

Can be save by a debugger for later use

Anatomy of a function call

```
int __stdcall foo(char * arg1, int arg2);

void bar()
{
   int my_val = 27;
   .
   int retval = foo("abcd", my_val);
   .
}
```

`__stdcall` ?

- A binary level contract between caller and callee
- All arguments are passed on the stack
  (none are passed in registers)
- Order of passing is right to left
- Callee unwinds the stack
- Return value (if any) in **EAX**
- Core registers are preserved by callee
- Other conventions are
  `__cdecl`, `__fastcall`, `__thiscall`

```
retval = foo("abcd", my_val);
```

compiler's view

```
mov     eax, DWORD PTR _my_val$[ebp]
push    eax
push    OFFSET FLAT:$SG171
call    ?foo@@YGHPADAAH@Z                  ; foo
mov     DWORD PTR _retval$[ebp], eax
```

```
c:\tmp>Cl /Fas /Od foo.cpp
```

output in foo.asm

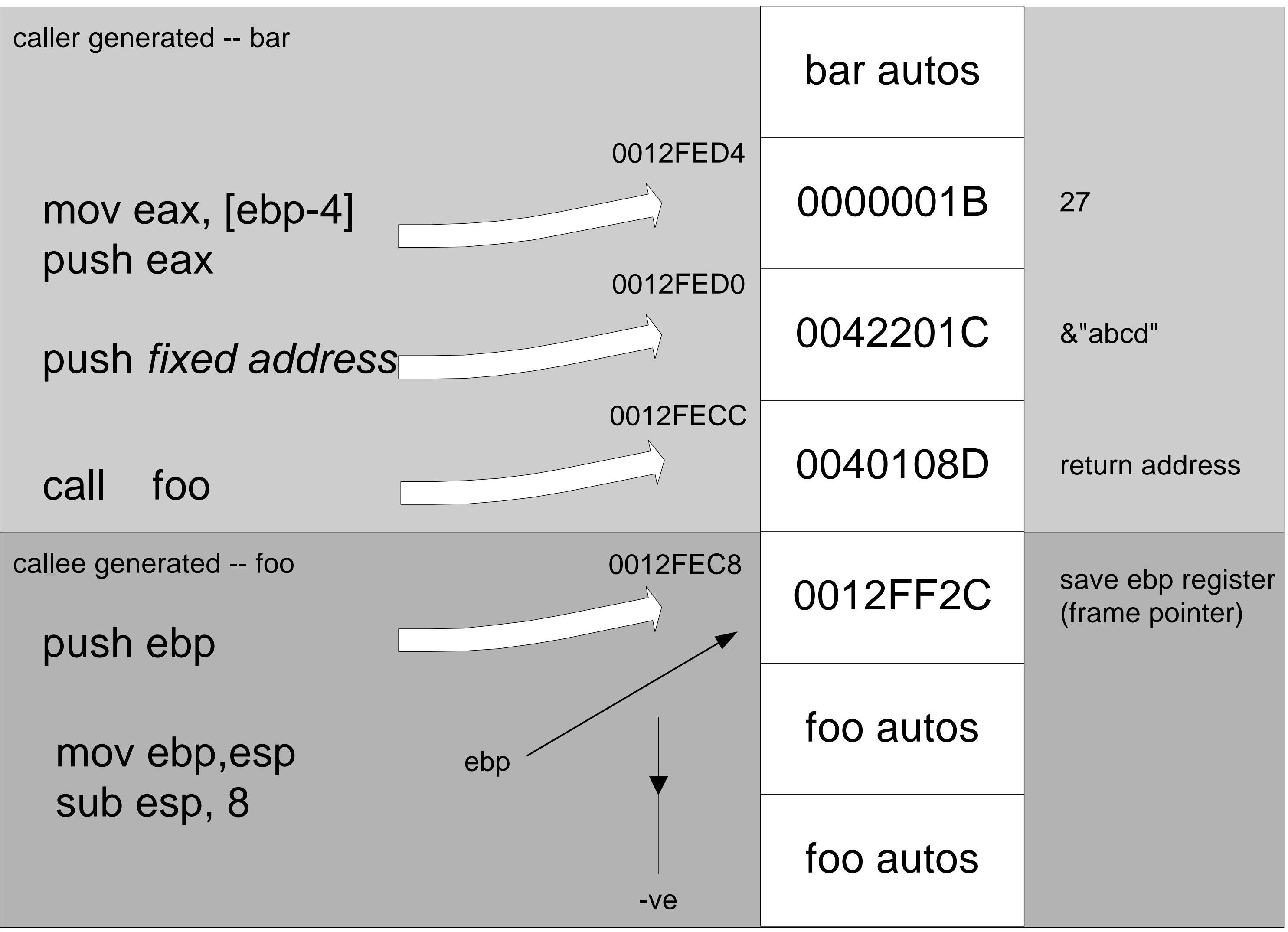
retval = foo(“abcd”, my_val);

debugger’s view

```
mov     eax, [ebp-4]                   // store value of my_val in eax
push    eax                            // push arg2 onto stack
push    offset string “abcd”           // push arg1 onto stack
call    foo                            // transfer control to foo
mov     DWORD PTR [ebp-8], eax         // save return value in an automatic
```

Stop on a break point
Show disassembler

Generated call stack



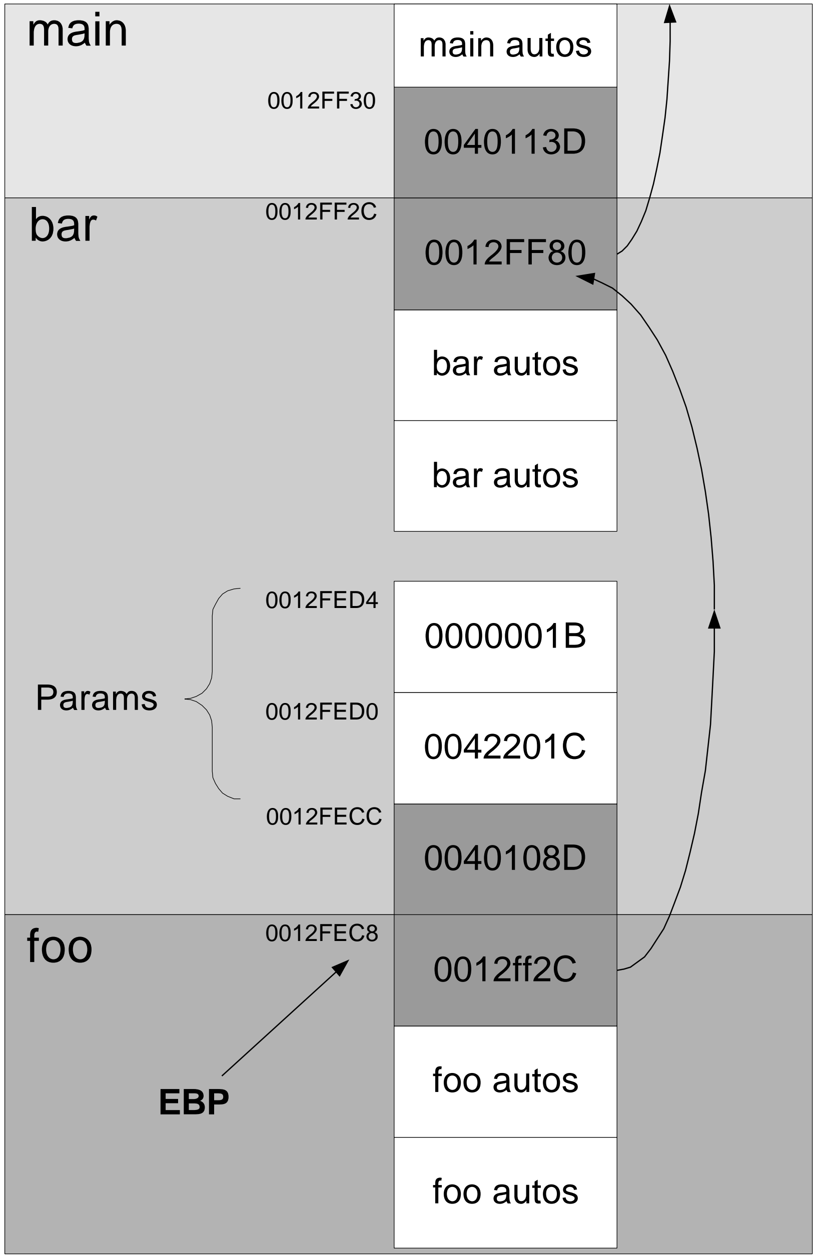
The big picture

What we have is a singly linked list of return addresses. With EBP pointing to the head

Above each node are function parameters

Below each node are automatics

Between is temporary space



Optimized code

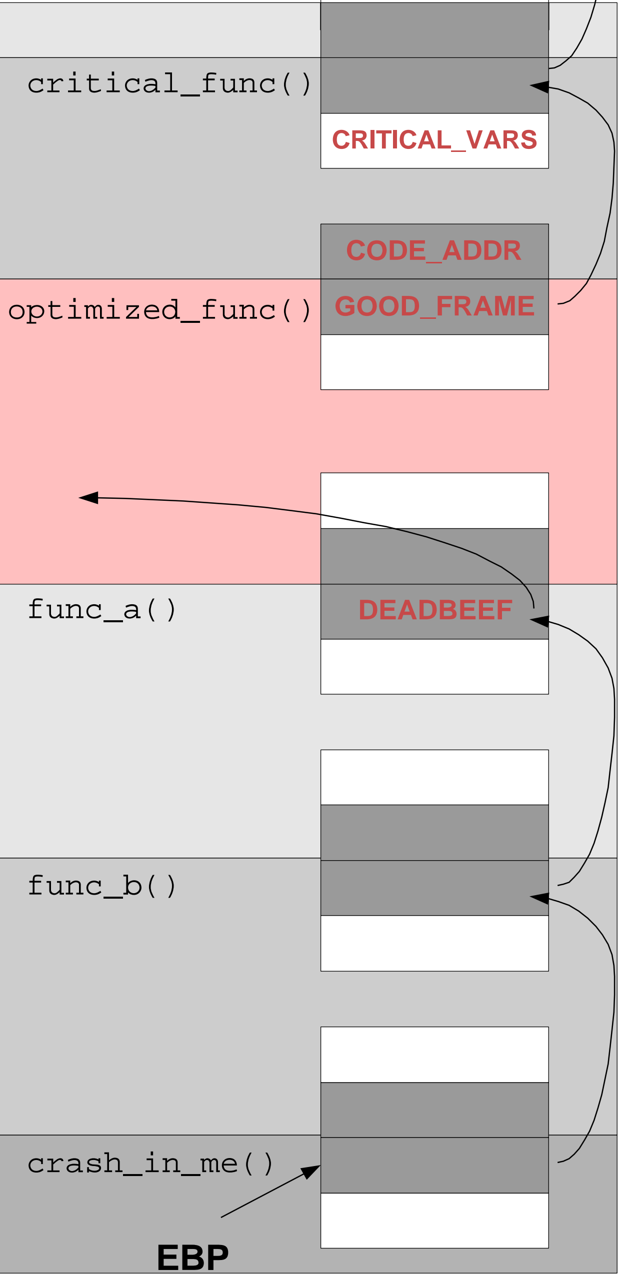
i386 cpu is CISC not RISC,
so registers are valuable and sparse.

EBP is often reused for other purposes

Result is a "fragmented" singly linked list

`critical()` context:

`EIP =` CODE_ADDR
`EBP =` GOOD_FRAME



Modules list

module	address
`program.exe`	`0x00400000 - 0x0042BFFF`
`your.dll`	`0x20020000 - 0x2002BFFF`
`kernel32.dll`	`0x77F00000 - 0x77F5EFFF`
`ntdll.dll`	`0x77F60000 - 0x77FBDFFF`

- Code plus global data
- Code usually precedes data
- Any code or data for program.exe will be at an address in the range 0x00400000 to 0x0042BFFF

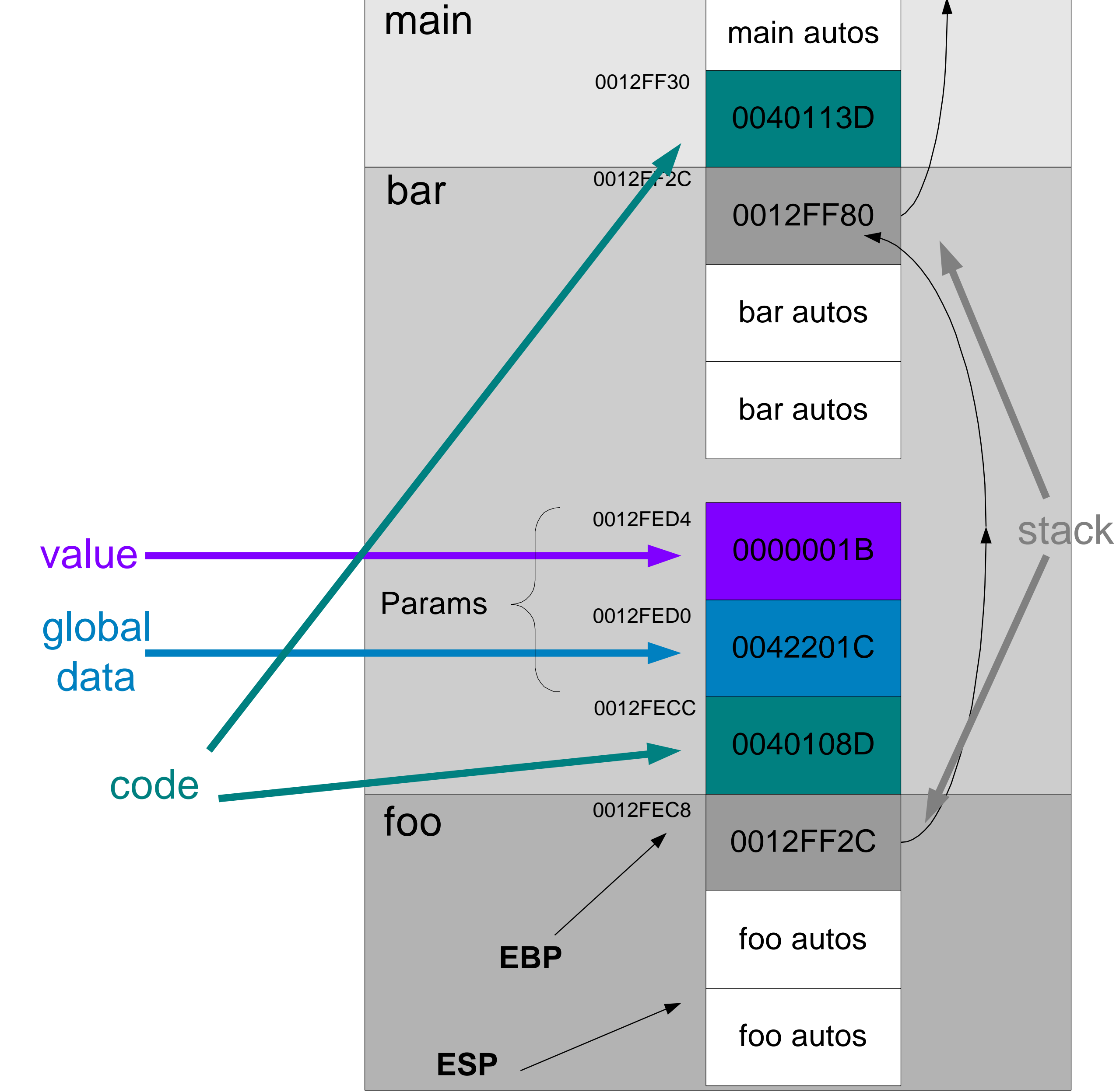
Tip: Always fix the "*base address*" of <u>your</u> dlls.

Tip: Examine module layout with **`dumpbin /headers`**

Recognizing values

If you can identify the code addresses, you can find the nodes in a fragmented singly linked list

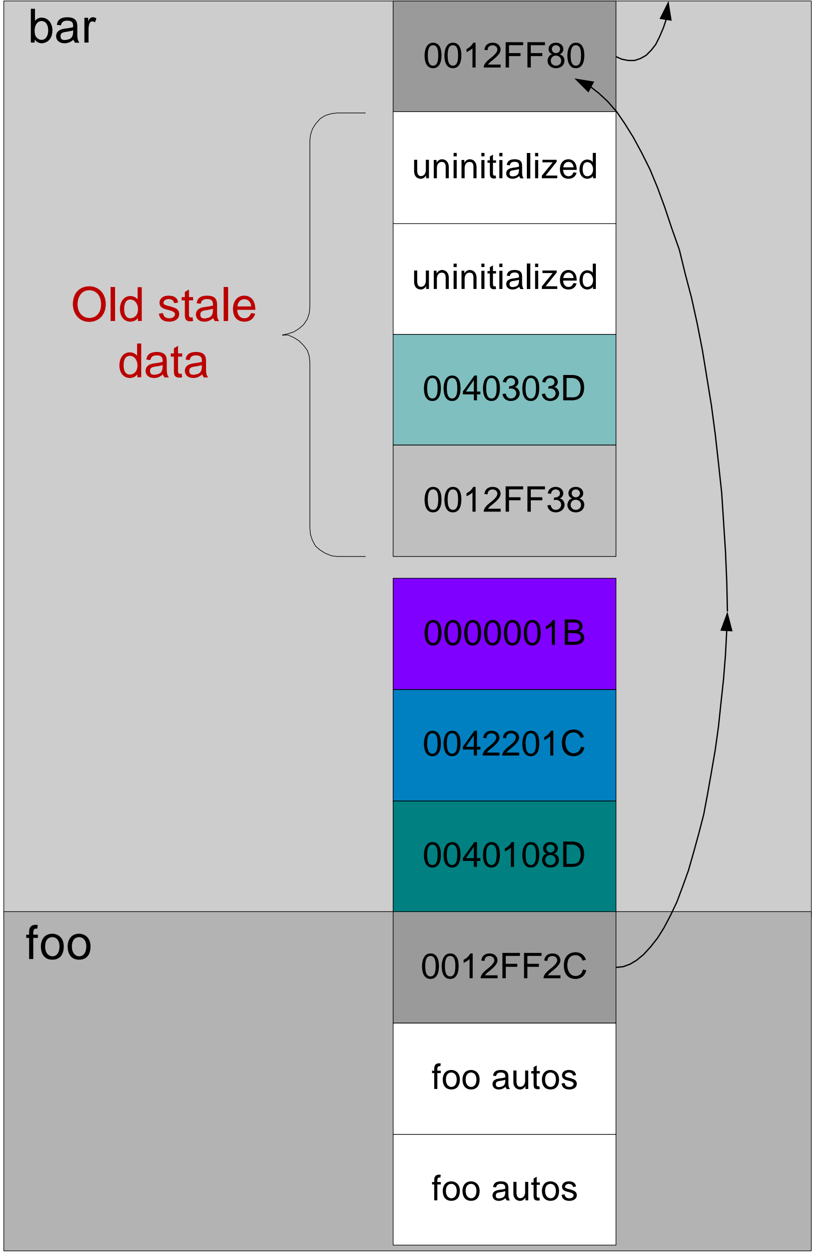
Code addresses always point to the instruction after a `call` instruction



Laying false trails

```
int bar()
{
   int array[100]; //uninitialized
                   // sub esp, 400
```

```
int bar()
{
   int array[100] = {0};
```



Application: stack overflow

- Find the beginning of the stack, take ESP add the default stack length
- Search memory backwards from this address, until the address of `func1` or `func2` appear (repeatedly)
- Find the previous code address on the stack
- Use the debugger to look up the function name (disassemble, evaluate or watch it)
- Check if it could call `func1` or `func2`
- Instruct debugger to decode stack frame or decode it by hand

Application: short stack trace

3rd party code

Your code?

```
ntdll!RtlRaiseException(unsigned int 0xC0000008)
kernel32!_CreateFileW@28
```

- The most common cause of a short stack trace is missing symbol files. Make sure all symbol files are loaded. Especially for the last module listed on the stack (`kernel32` in this case)
- Find the last decoded code address
- Examine memory for the next potential code address
- Verify the code address. Disassemble the code at that location it should follow a "`call`" instruction
- Lookup the symbol
- Read the source code

Application: no stack trace

- EIP is corrupt, thus the crash, and EBP is probably useless
- Take ESP find the first potential code address on the stack
- Set the function context manually:
  set EIP to this value
  set EBP to the position of this code on the stack -4
- Instruct the debugger to redisplay the call stack

Summary

- Symbolic debuggers need symbols!
- Use a symbol server and symbol cache
- Get familiar with more than one debugger
- Use mini dumps
- Code locations are readily identifiable on the stack

Links and further reading

John Robbins, Debugging Applications, Microsoft Press. Also his Bug Slayer columns MSJ/MSDN magazine.
Various MSJ/MSDN magazine articles by Matt Pietrek
How debuggers work, Rosenberg, Wiley
www.microsoft.com/ddk/debugging news: microsoft.public.windbg microsoft.public.vc.debugger
www.sysinternals.com
www.bugbrowser.com