Defending Computer Networks
Lecture 4: Exploit Defenses

Stuart Staniford

Adjunct Professor of Computer Science

Logistics

- HW1
  - http://www.cs.cornell.edu/courses/cs5434/2014fa/hw1.pdf
- Virtual Machine
  - http://www.cs.cornell.edu/courses/cs5434/2014fa/VirtualMachines.zip
- Bad reading links fixed

Latest News

Data Breaches — 112 comments

03 Data: Nearly All U.S. Home Depot Stores Hit

SEP 14

New data gathered from the cybercrime underground suggests that the apparent credit and debit card breach at **Home Depot** involves nearly all of the company's stores across the nation.

Evidence that a major U.S. retailer had been hacked and was leaking card data first surfaced Monday on the cybercrime store **rescator[dot]cc**, the shop that was principally responsible for selling cards stolen in the **Target**, **Sally Beauty**, **P.F. Chang's** and Harbor Freight credit card breaches.

> Here's the kicker: *A comparison of the ZIP code data between the unique ZIPs represented on Rescator's site, and those of the Home Depot stores shows a staggering 99.4 percent overlap.*

http://krebsonsecurity.com/

In other good news…

Cyber Attack Nets 4.5 Million Records From Large Hospital System

+ Comment Now + Follow Comments

In what could well be the largest single health data breach by a publicly traded company, Community Health Systems (CHS) announced earlier today that information on 4.5 million patients was stolen as part of a cyber attack they believe originated in China.

> *In July 2014, Community Health Systems, Inc. (the "Company") confirmed that its computer network was the target of an external, criminal cyber attack that the Company believes occurred in April and June, 2014. The Company and its forensic expert, Mandiant (a FireEye Company), believe the attacker was an "Advanced Persistent Threat" group originating from China who used highly sophisticated malware and technology to attack the Company's systems.* SEC Form 8–K as filed by CHS on 8/18/2014

Main Goals for Today

- Heap/BSS Overflows and Vulnerabilities
  - Just a little taste
- Stack Canary Defenses
- Understand NX defenses against overflows
  - And sketch return oriented programming
- Understand Address Space Randomization
  - And how the dark side can work around it
- Discussion today probably a little sketchier
  - Want to point to various issues
  - But need to move on to other topics

Heap/BSS Overflows

- Heap is app dynamically allocated memory
  - malloc/new
- BSS is segment for static/global variables.
- Code vulnerabilities are conceptually similar to in stack case, but with heap/bss variables

```
char* foo = (char*)malloc(64);
if(foo)
  strcpy(foo, user)
```

Simple BSS example

```
int main(int argc, char **argv)
{
  FILE *tmpfd;
  static char buf[BUFSIZE], *tmpfile;
  tmpfile = "/tmp/vulprog.tmp";
  printf("Enter one line of data to put in %s: ", tmpfile);
  gets(buf);
  tmpfd = fopen(tmpfile, "w");
  fputs(buf, tmpfd);
  fclose(tmpfd);
}
```

Adapted from http://netsec.cs.northwestern.edu/media/readings/heap_overflows.pdf

Simple heap example

```
struct myObject
{
  char name[64];
  int (*foo)(int);
 …
}
```

Note that in C++, virtual functions are stored implicitly in object structure as function pointers

Use After Free()

CWE-416: Use After Free

Use After Free

Weakness ID: 416 *(Weakness Base)* **Status:** Draft

Description

Description Summary

Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.

Extended Description

The use of previously-freed memory can have any number of adverse consequences, ranging from the corruption of valid data to the execution of arbitrary code, depending on the instantiation and timing of the flaw. The simplest way data corruption may occur involves the system's reuse of the freed memory. Use-after-free errors have two common and sometimes overlapping causes:

- Error conditions and other exceptional circumstances.
- Confusion over which part of the program is responsible for freeing the memory.

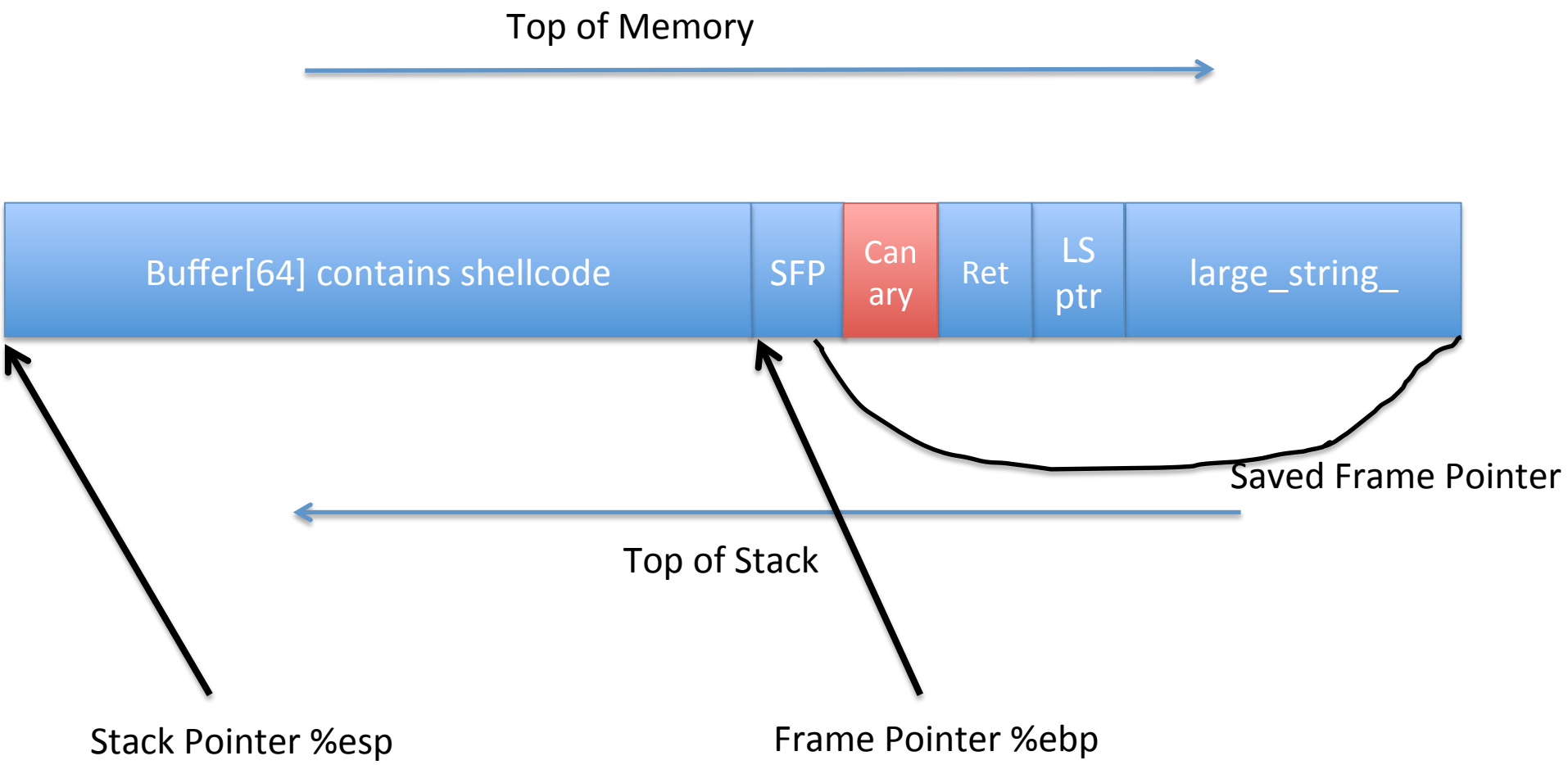
In this scenario, the memory in question is allocated to another pointer validly at some point after it has been freed. The original pointer to the freed memory is used again and points to somewhere within the new allocation. As the data is changed, it corrupts the validly used memory; this induces undefined behavior in the process.

If the newly allocated data chances to hold a class, in C++ for example, various function pointers may be scattered within the heap data. If one of these function pointers is overwritten with an address to valid shellcode, execution of arbitrary code can be achieved.

Heap Issues in General

- Often exploitable
- Not nearly as cookie-cutter as stack issues
- Requires more code-analysis from the attacker
  - Each case is different

Stack Defense Canaries

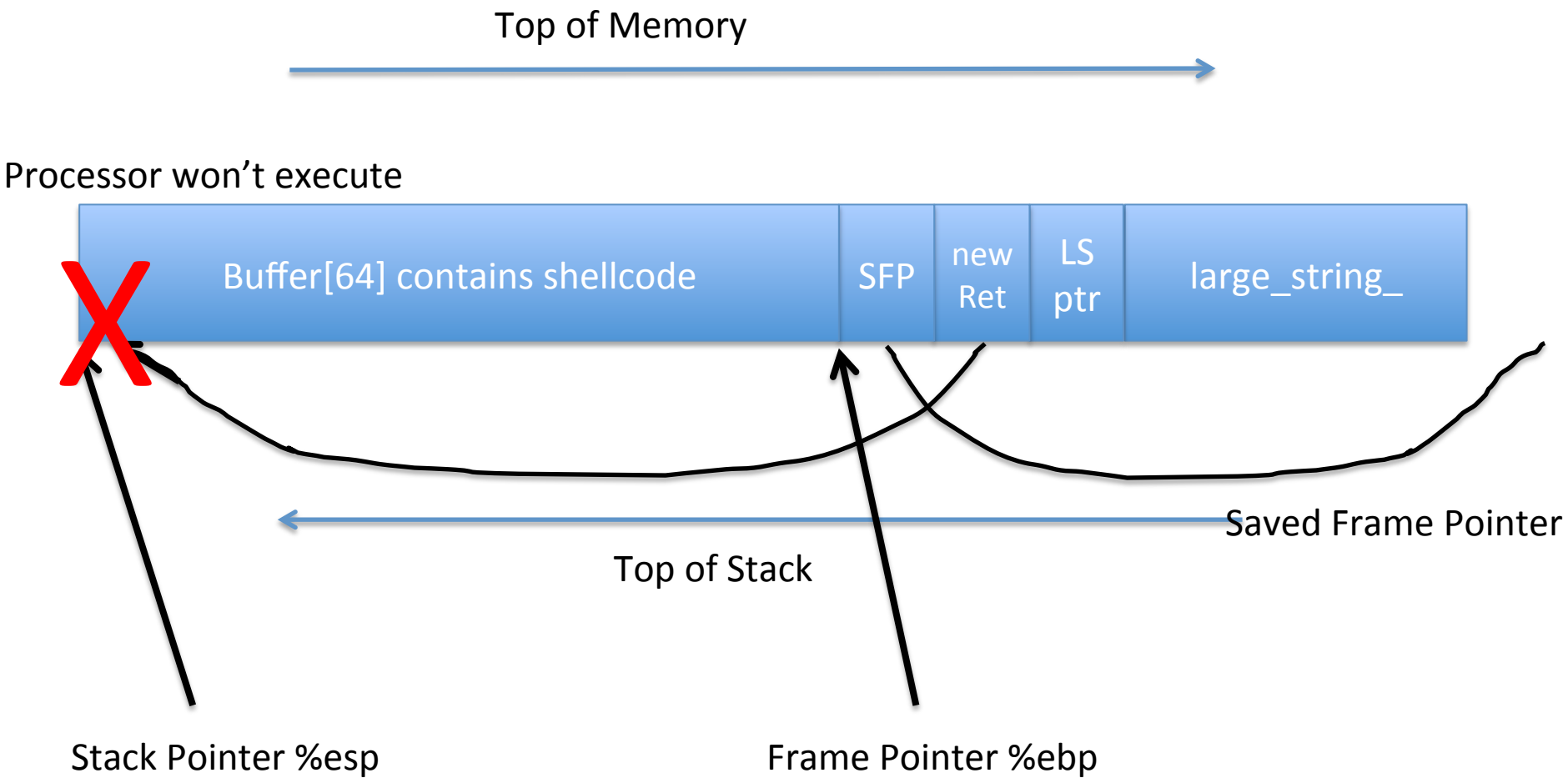


Not invincible. Eg http://phrack.org/issues.html?issue=56&id=5

NX/DEP/W^X

- Related mechanisms with common theme
  - Let's not execute the stack/heap
  - That way, cannot inject shellcode into buffer
  - And then point RIP at buffer contents
  - Ditto in format string attack, cannot put shellcode into buffer
- Requires hardware/OS support
  - But we have that now

NX/DEP/W^X


Top of Memory
Processor won't execute
Buffer[64] contains shellcode
SFP
new Ret
LS ptr
large_string_
Saved Frame Pointer
Top of Stack
Stack Pointer %esp
Frame Pointer %ebp

NX Bit

- General term for hardware feature
  - Originally AMD term
  - XD Bit (Intel)
  - XN Bit (ARM)
- Implemented in the page table
  - Bit 63 says this page cannot be executed
  - Hardware will enforce when doing memory lookup on instruction pointers in virtual address space
  - OS needs to manage the bits on the pages
    - Make sure stack and heap pages cannot execute

DEP: Data Execution Prevention

- Term for the OS Level feature
- Particularly on MS Windows
  - Controllable on a process-by-process basis
  - First optionally available on XP SP2 (circa 2004)
  - Default is still only to be available on core OS stuff
  - Optionally turn on for everything
    - Breaks some applications

W^X

- Write XOR Execute
- Extended version of idea
- All virtual pages can be
  - either writeable or executable
  - But not both
- Prevents self-modifying code
- OpenBSD, OS X, some Linux have full W^X

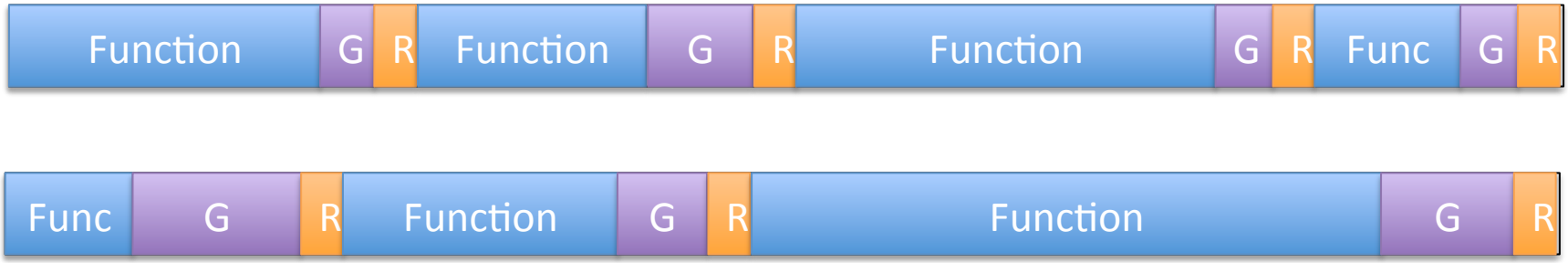
Defeating NX: Return to LIBC

- We can't make RIP point to our code
- But we can make it point to any pre-existing code on system
  - In text segment, so executable
- And we can set up stack beforehand
- Eg call system();
- So NX not invincible by itself

ROP: Return Oriented Programming

- Generalization of return-to-libc idea
- Shacham, 2007
- Idea is to crawl libc (etc), and find a series of "gadgets"
- A gadget is a useful bit of code right before the ret instruction of some function
- Might just be one or two instructions

Libc from a ROP POV



Etc, etc…

More ROP

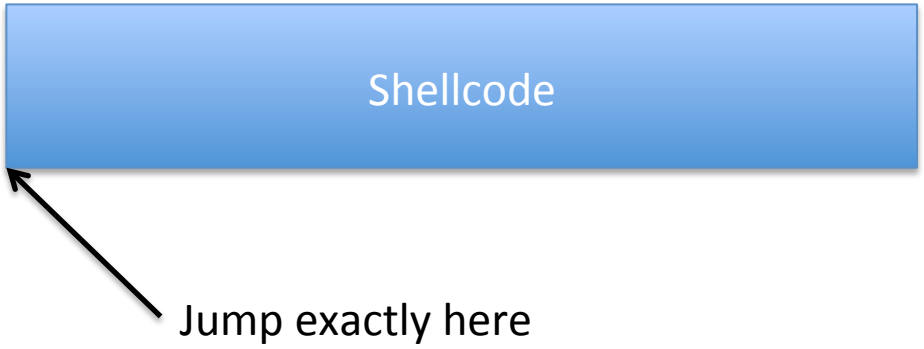
- Then call a bunch of these in sequence
  - from the (scribbled on) stack
- Shachem showed that libc ROP gadgets form a general purpose computation framework that can do anything.
- Very hard to fix this by surgery on libc

NOP Sleds

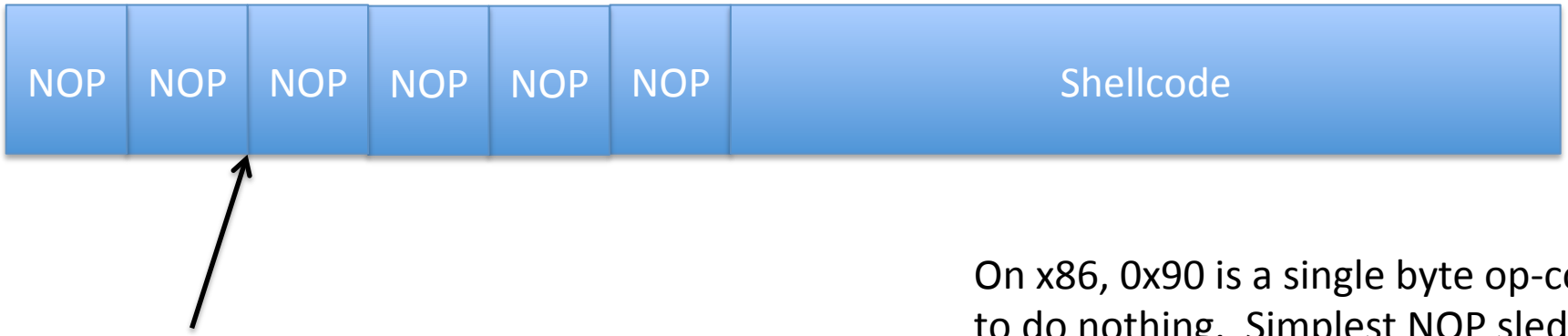
- When we overwrite an address in memory
  - Say a RIP
- We need to know what value to put.
- Simple case:
  - Beginning of shellcode in buffer
- But this is fragile.
  - Any slight difference in code version,
  - Even if it didn't affect the vulnerability
  - Could change the jump address

So allow for some imprecision

Instead of



Do



Jump somewhere in here. Now our exploit is less fragile. Assuming we have the space.

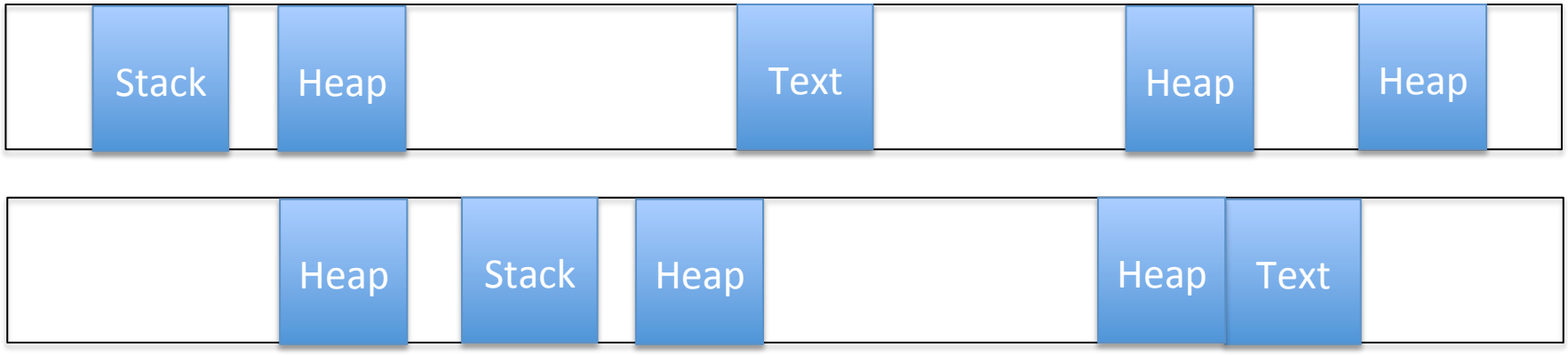
On x86, 0x90 is a single byte op-code to do nothing. Simplest NOP sled.

Address Space Layout Randomization

Basic insight is to make it really hard to figure out what address to jump to. Put key parts of the program in random places in memory

Instead of loading program into memory the same way every time:

Randomize:



So Now

- When we overwrite RIP, we don't know where to point, not even close
  - If we could get close, could use a NOP-sled

ASLR

- Now available on all major OS's
- Not all legacy code is compiled this way
  - May not be position-independent
- But increasingly becoming standard
- So a fully modern exploit must get past
  - Canaries
  - NX/DEP
  - ASLR
- All at the same time…
- But let's look at ASLR in isolation for a moment

Brute Forcing ASLR

- Loading at run time, we can't do fine-grained randomization
  - Code, for example, has all kinds of internal jumps that must be known, so code can't easily be jumbled up at the micro scale.
- Limited to moving around big chunks (stack, text, etc)
  - Consider an 8MB stack in 4GB (32 bit machine)
  - 512 possible positions. Not outrageous to guess
  - Much more difficult on 64 bit machines

Leaking Addresses

- Anything that allows us to see an address,
  - lets us get a handle on where that kind of thing lives
  - Eg format string vulnerability allows us to inspect the stack before doing our attack
    - We can quickly figure out where everything lives
      - Text pointers in RIPs
      - Stack pointers in stack frame bases
      - Heap pointers in local variable pointers to heap buffers

Defeating ALSR/DEP combined

- Any non-ALSR code can be analyzed for ROP.
  - Still sometimes libraries/code lying around. Eg
    - https://blogs.technet.com/b/srd/archive/2013/08/12/mitigating-the-ldrhotpatchroutine-dep-aslr-bypass-with-ms13-063.aspx

The bypass takes advantage of a predictable memory region known as SharedUserData that exists at a fixed location (0x7ffe0000) in every process on every supported version of Windows. On 64-bit versions of Windows prior to Windows 8, this region contains pointers to multiple functions in the 32-bit version of NTDLL that is used by WOW64 processes as shown below:

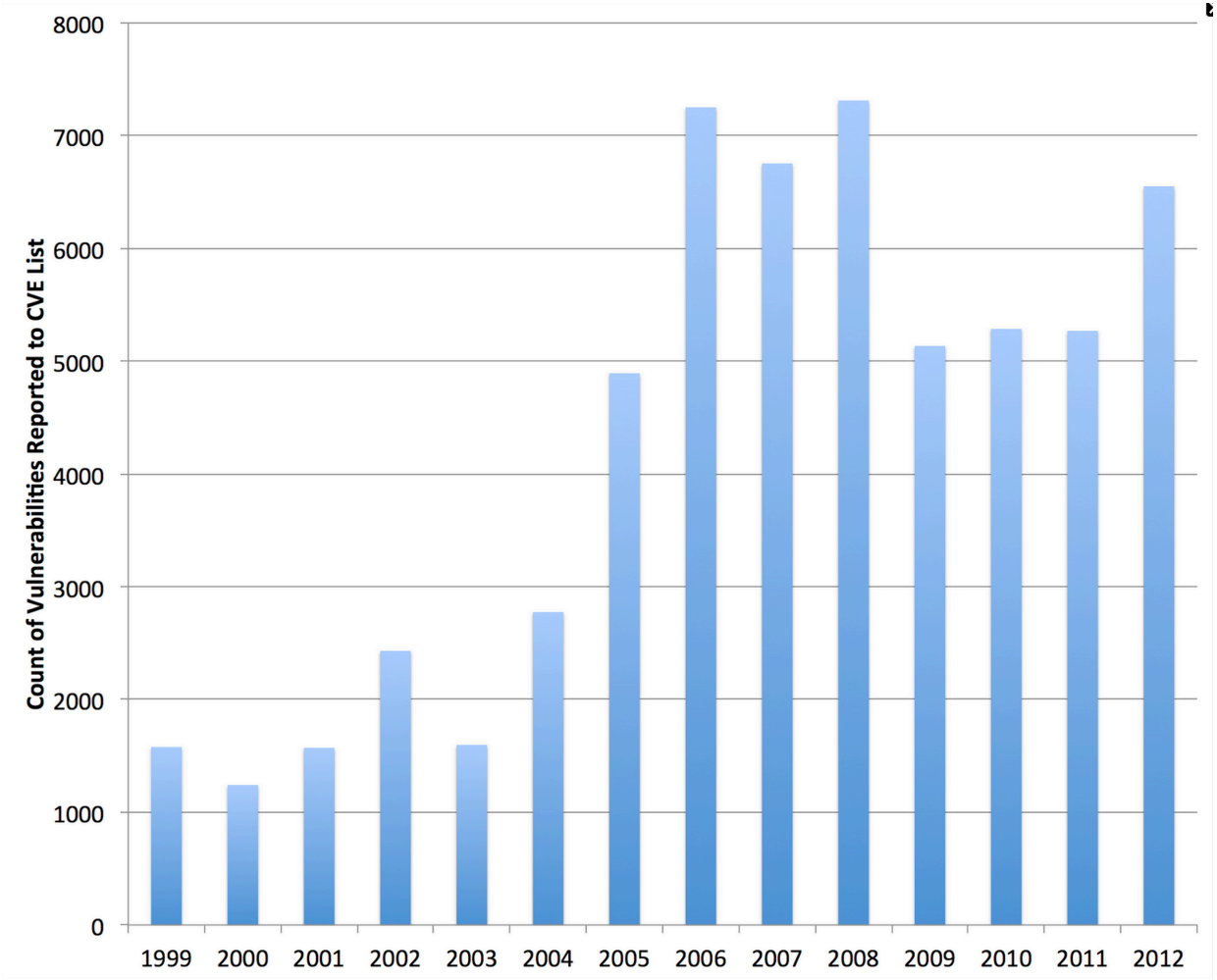
```
0:000> dds 7ffe0340 Lc
00000000`7ffe0340  77829ce9 ntdll32!LdrInitializeThunk
00000000`7ffe0344  77800100 ntdll32!KiUserExceptionDispatcher
00000000`7ffe0348  77800028 ntdll32!KiUserApcDispatcher
00000000`7ffe034c  778000b8 ntdll32!KiUserCallbackDispatcher
00000000`7ffe0350  7788f8d4 ntdll32!LdrHotPatchRoutine
00000000`7ffe0354  77822551 ntdll32!ExpInterlockedPopEntrySListFault
00000000`7ffe0358  7782251b ntdll32!ExpInterlockedPopEntrySListResume
00000000`7ffe035c  77822553 ntdll32!ExpInterlockedPopEntrySListEnd
00000000`7ffe0360  77800190 ntdll32!RtlUserThreadStart
00000000`7ffe0364  77892dfd ntdll32!RtlpQueryProcessDebugInformationRemote
00000000`7ffe0368  778517d9 ntdll32!EtwpNotificationThread
00000000`7ffe036c  777f0000 ntdll32!CsrServerApiRoutine
```

Defeating ALSR/DEP

- Getting harder – Microsoft will pay $100k for any new methods of doing it on Windows
  - http://www.microsoft.com/security/msrc/report/bypass_bounty.aspx

Bottom Line: Defenses Help

But not a panacea yet:



Problem Still Not Fully Solved

Microsoft » Windows 8 : Vulnerability Statistics

Vulnerabilities (**88**) CVSS Scores Report Browse all versions Possible matches for this product Related Metasploit Modules

Related OVAL Definitions : Vulnerabilities (78) Patches (0) Inventory Definitions (2) Compliance Definitions (0)

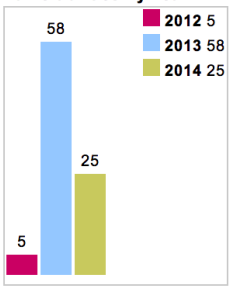
Vulnerability Feeds & Widgets

Vulnerability Trends Over Time

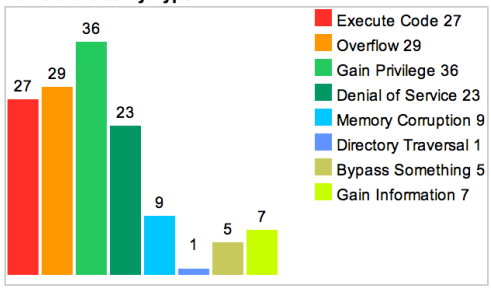
Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
2012	5		3	2								2			
2013	58	17	18	22	6			1		2	3	25			4
2014	25	6	6	5	3					3	4	9			
Total	88	23	27	29	9			1		5	7	36			4
% Of All		26.1	30.7	33.0	10.2	0.0	0.0	1.1	0.0	5.7	8.0	40.9	0.0	0.0	

Warning : Vulnerabilities with publish dates before 1999 are not included in this table and chart. (Because there are not many of them and they make the page look bad; and they may not be actually published in those years.)

Vulnerabilities By Year



Vulnerabilities By Type



http://www.cvedetails.com/product/22318/Microsoft-Windows-8.html