Lecture 04 – Control Flow (cont)

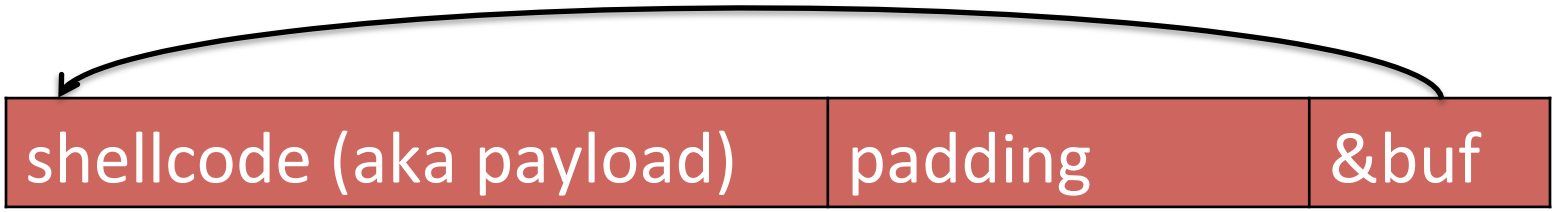
Michael Bailey
University of Illinois
ECE 422/CS 461 – Spring 2018

Major portions from
David Brumley
Carnegie Mellon University

Control Flow Hijack Defenses
Canaries, DEP, and ASLR

David Brumley
Carnegie Mellon University

Control Flow Hijack: Always control + computation



computation + ***control***

- code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- …

Same principle, different mechanism

Control Flow Hijacks

… happen when an attacker gains control of ***the instruction pointer****.*

Two common hijack methods:

- buffer overflows
- format string attacks

Control Flow Hijack Defenses

Bugs are the root cause of hijacks!

- Find bugs with analysis tools
- Prove program correctness

Mitigation Techniques:

- Canaries
- Data Execution Prevention/No eXecute
- Address Space Layout Randomization

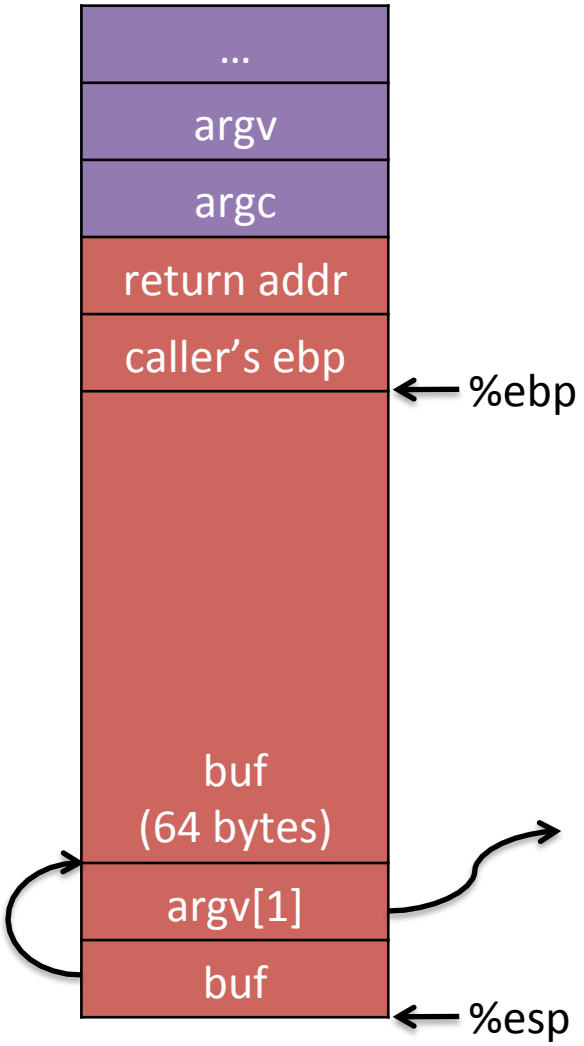
CANARY / STACK COOKIES

http://en.wikipedia.org/wiki/File:Domestic_Canary_-_Serinus_canaria.jpg

“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

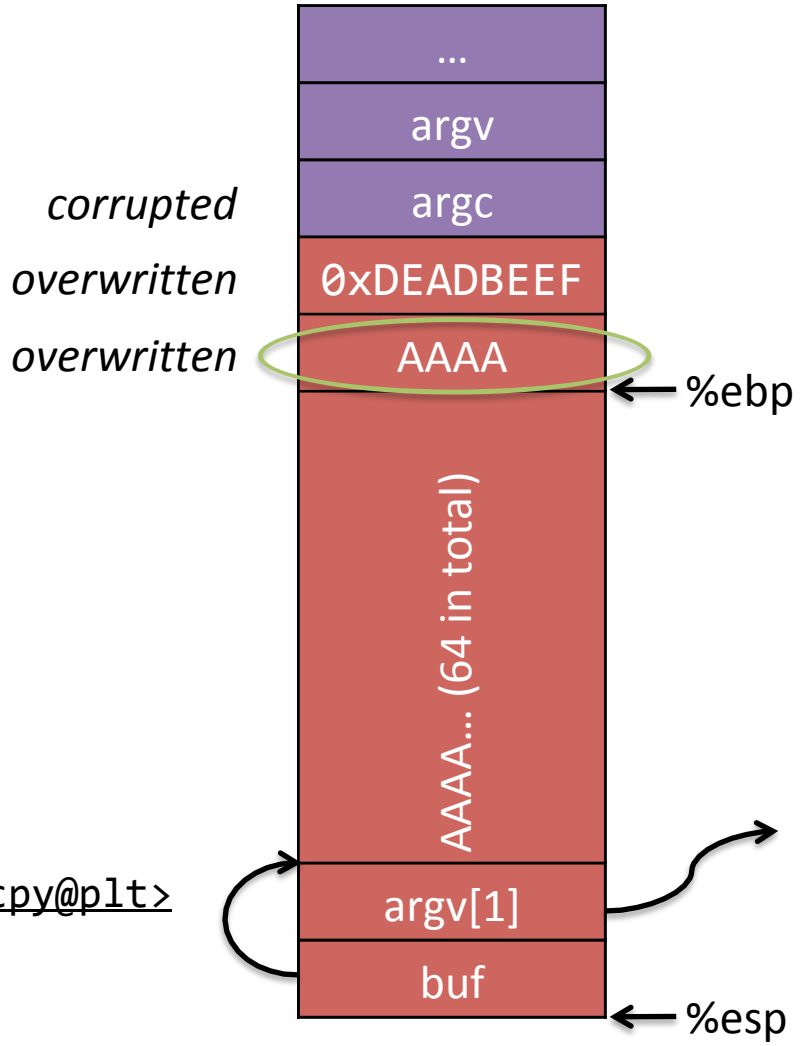


7

“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



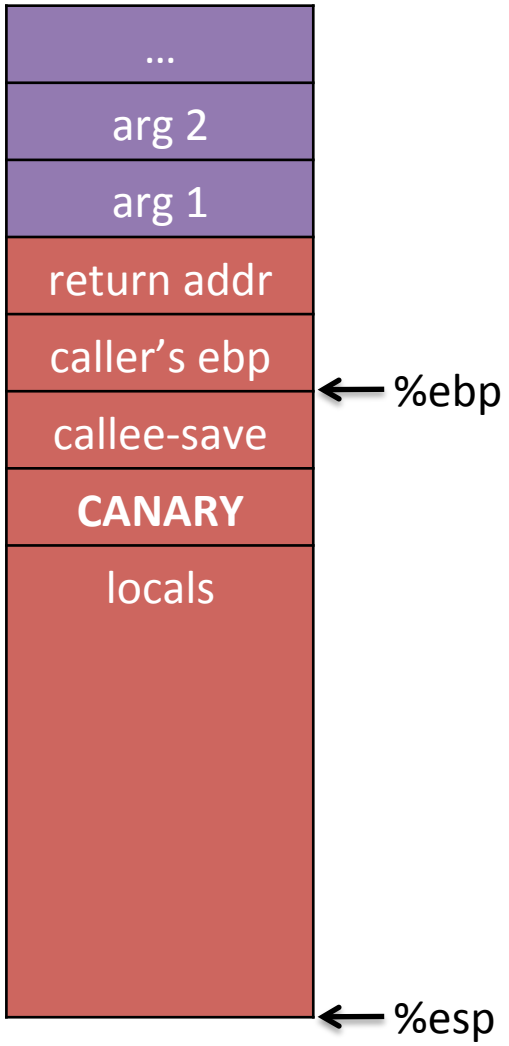
8

StackGuard[Cowen etal. 1998]

Idea:

- prologue introduces a ***canary word*** between return addr and locals
- epilogue checks canary before function returns

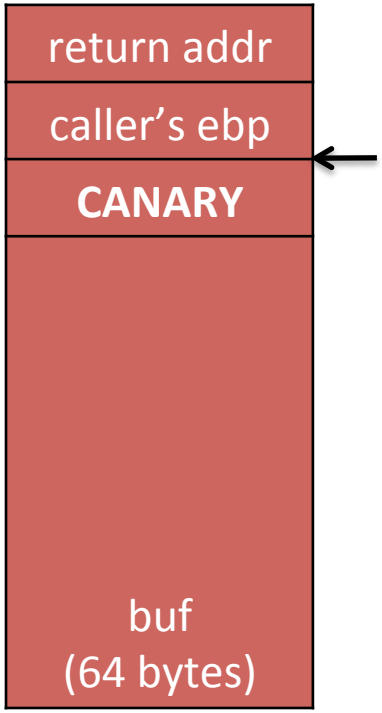
Wrong Canary => Overflow



gcc Stack-Smashing Protector (ProPolice)

```
Dump of assembler code for function main:
   0x08048440 <+0>:  push   %ebp
   0x08048441 <+1>:  mov    %esp,%ebp
   0x08048443 <+3>:  sub    $76,%esp
   0x08048446 <+6>:  mov    %gs:20,%eax
   0x0804844c <+12>: mov    %eax,-4(%ebp)
   0x0804844f <+15>: xor    %eax,%eax
   0x08048451 <+17>: mov    12(%ebp),%eax
   0x08048454 <+20>: mov    4(%eax),%eax
   0x08048457 <+23>: mov    %eax,4(%esp)
   0x0804845b <+27>: lea    -68(%ebp),%eax
   0x0804845e <+30>: mov    %eax,(%esp)
   0x08048461 <+33>: call   0x8048350 <strcpy@plt>
   0x08048466 <+38>: mov    -4(%ebp),%edx
   0x08048469 <+41>: xor    %gs:20,%edx
   0x08048470 <+48>: je     0x8048477 <main+55>
   0x08048472 <+50>: call   0x8048340 <__stack_chk_fail@plt>
   0x08048477 <+55>: leave
   0x08048478 <+56>: ret
```

Compiled with v4.6.1:
`gcc -fstack-protector -O1 …`



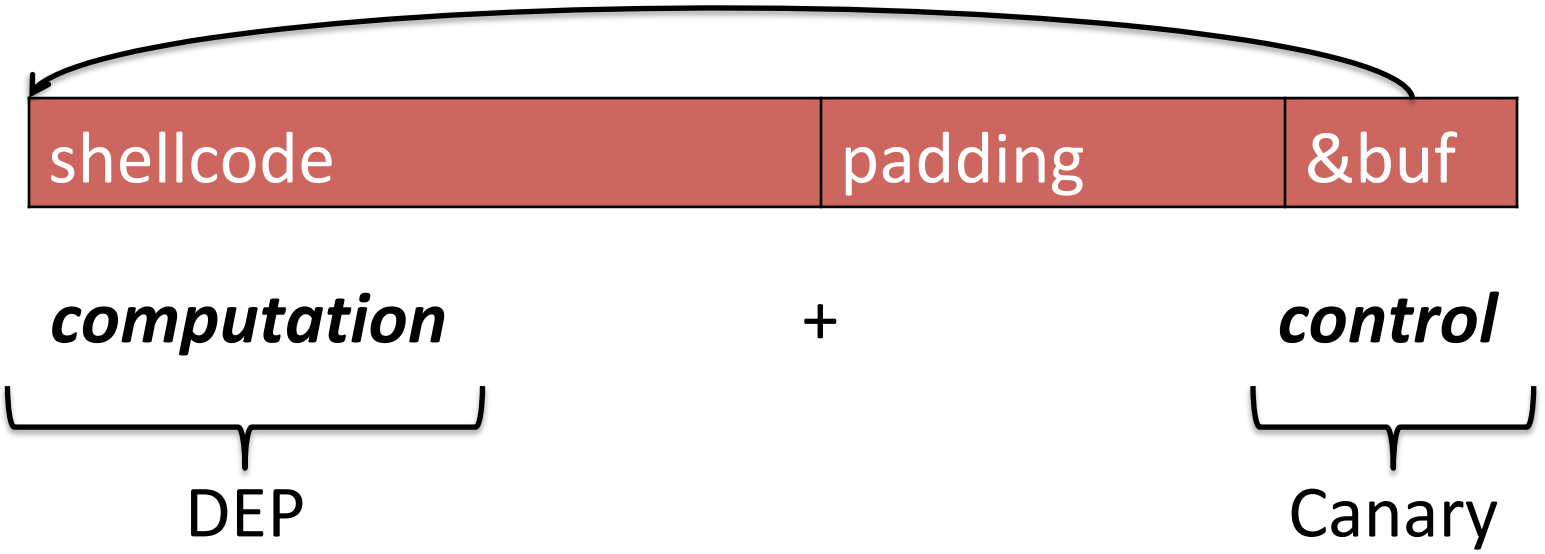
10

Canary should be **HARD** to Forge

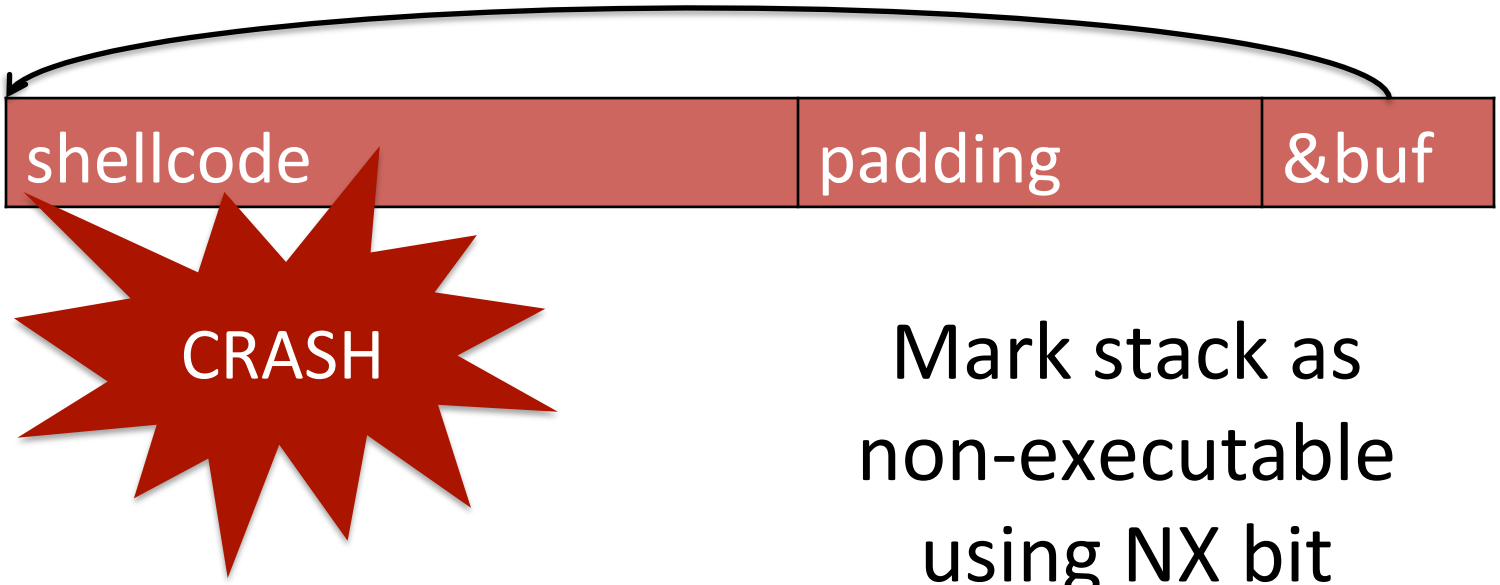
- Terminator Canary
  - 4 bytes: 0,CR,LF,-1 (low->high)
  - terminate `strcpy()`, `gets()`, …
- Random Canary
  - 4 random bytes chosen at load time
  - stored in a guarded page
  - need good randomness

DATA EXECUTION PREVENTION (DEP) / NO EXECUTE (NX)

How to defeat exploits?



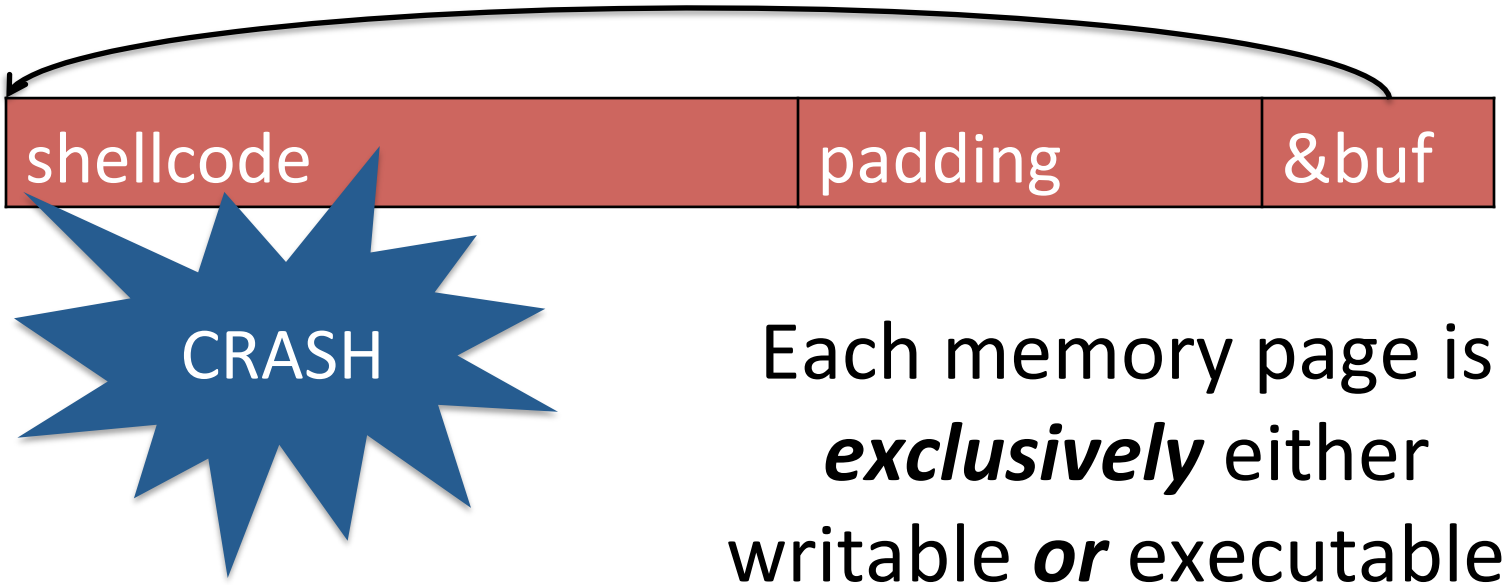
Data Execution Prevention



Mark stack as non-executable using NX bit

(still a Denial-of-Service attack!)

W ^ X



Each memory page is ***exclusively*** either writable ***or*** executable.

(still a Denial-of-Service attack!)

AMD, Intel put antivirus tech into chips

The companies plan to soon release technology that will allow processors to stop many computer attacks before they occur.

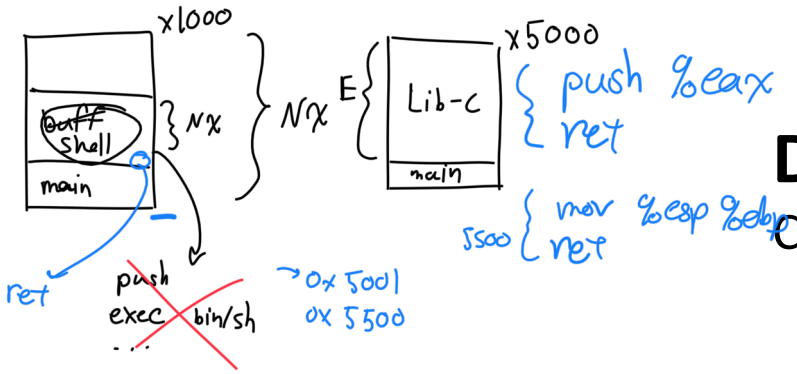
By Michael Kanellos | January 8, 2004 -- 23:22 GMT (15:22 PST) | Topic: Intel

LAS VEGAS--Advanced Micro Devices and Intel plan to soon release technology that will allow processors to stop many attacks before they occur.

Execution Protection by AMD, technology contained in AMD's Athlon 64 chips, prevents a buffer overflow, a common method used to attack computers. A buffer overflow essentially overwhelms a computer's defense systems and then inserts a malicious program in memory that the processor subsequently executes.

With Execution Protection, data in the buffer can only be read and, therefore, is prevented from doing its dirty work, John Morris, director of marketing at AMD, said in an interview Thursday at the Consumer Electronics Show here.

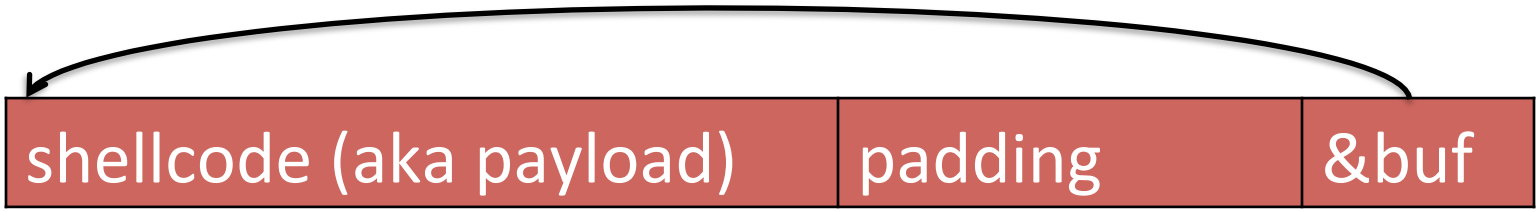
return-Oriented Programming



David Brumley
Carnegie Mellon University

Credit: Some slides from Ed Schwartz

Control Flow Hijack: Always control + computation

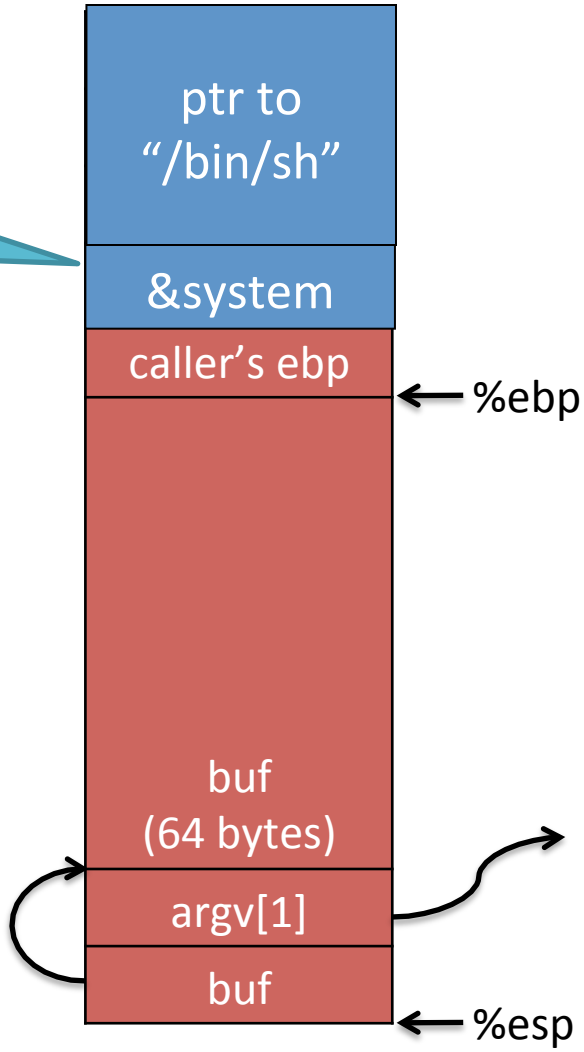


computation + ***control***

Return-oriented programming (ROP): shellcode without code injection

Motivation: Return-to-libc Attack

ret transfers control to `system`, which finds arguments on stack



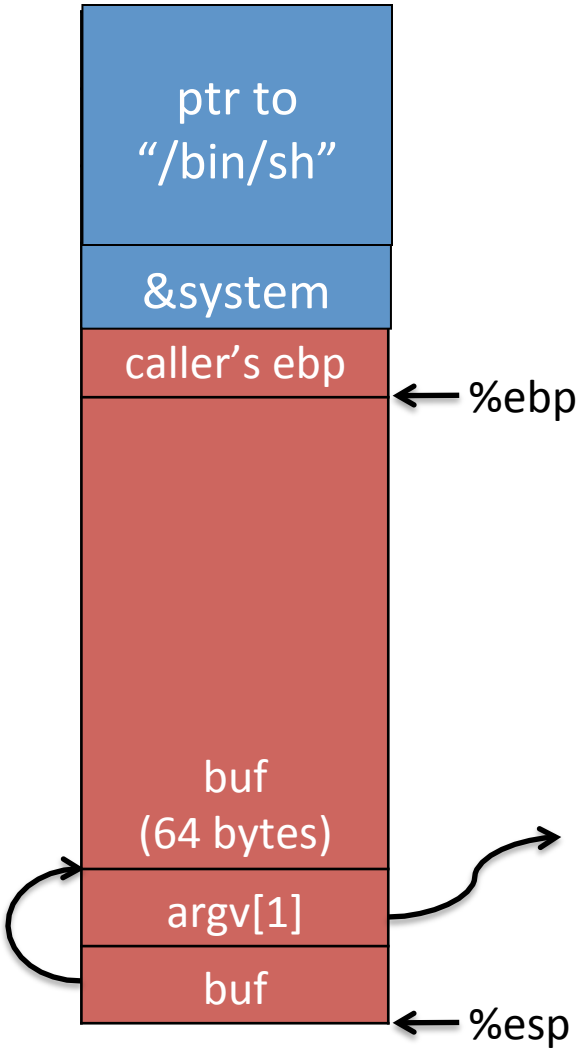
Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

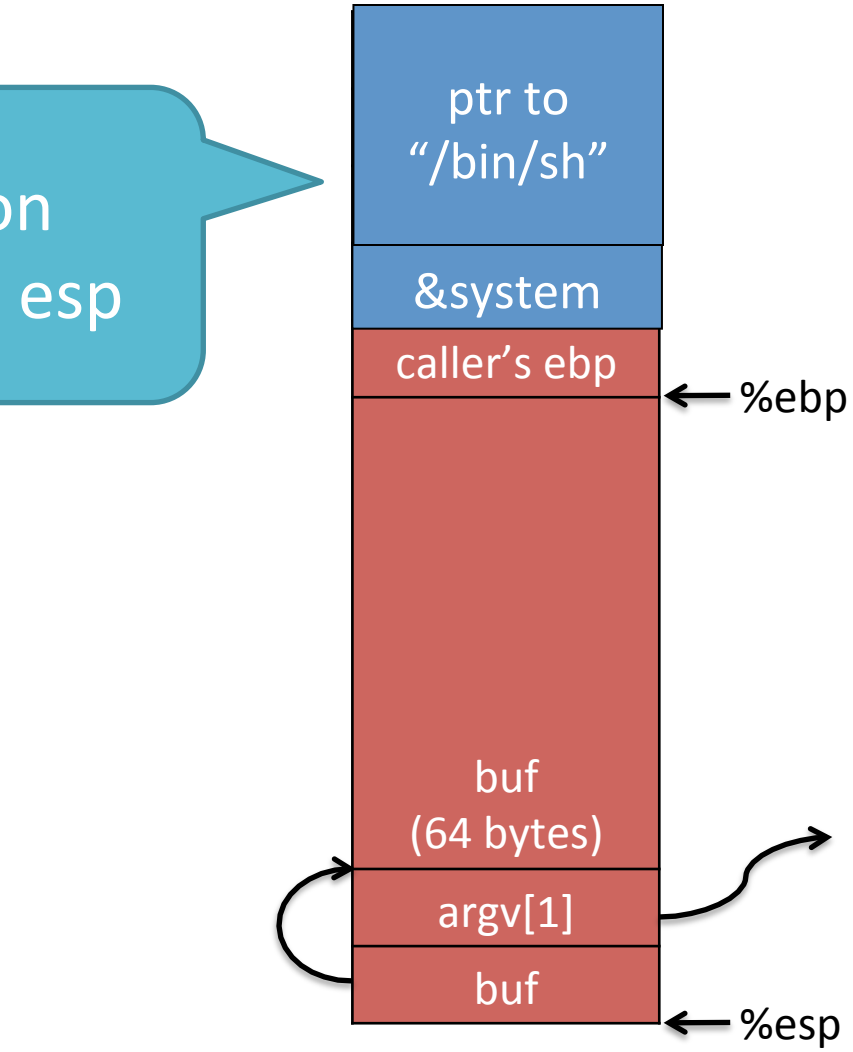
No injected code!

What if we don't know the absolute address any pointers to "/bin/sh"

(objdump gives addresses, but we don't know ASLR constants)



Need to find an instruction sequence, aka *gadget*, with esp



Scorecard for ret2libc

- No injected code → DEP ineffective
- Requires knowing address of system
- ... or does it.

RETURN ORIENTED PROGRAMMING TECHNIQUES

Geometry of Flesh on the Bone, Shacham et al, CCS 2007

ROP Programming

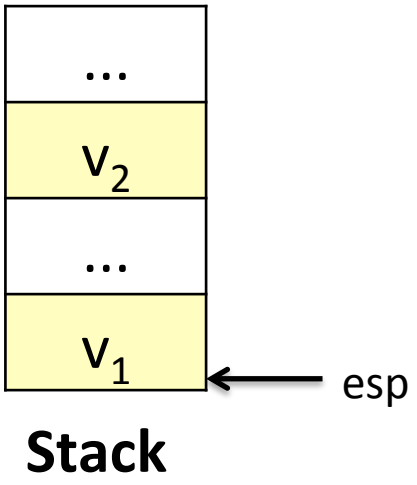
1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode

THERE ARE MANY
SEMANTICALLY EQUIVALENT
WAYS TO ACHIEVE THE SAME NET SHELLCODE EFFECT

Equivalence

```
Mem[v2] = v1
```

Desired Logic




```
a1: mov eax, [esp]
a2: mov ebx, [esp+8]
a3: mov [ebx], eax
```

Implementation 1

Gadgets

A gadget is any instruction sequence ending with `ret`

Image by Dino Dai Zovi

ROP Overview

- Idea: We forge shell code out of existing application logic gadgets

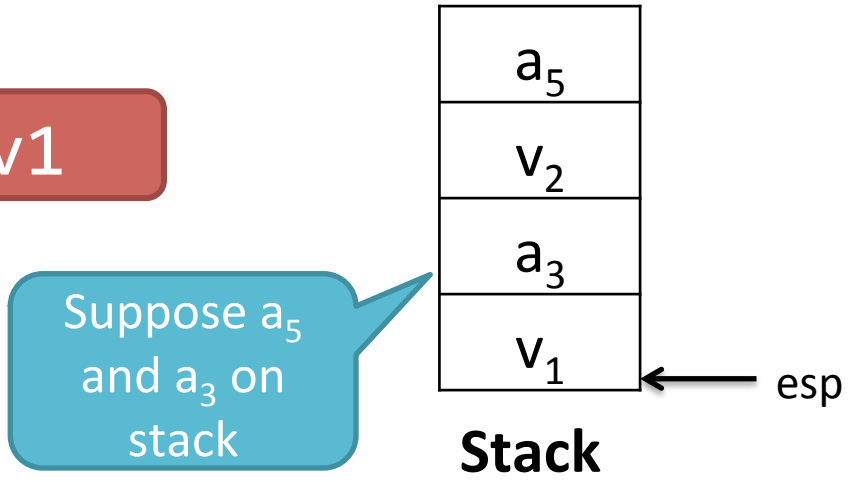
- Requirements:
  vulnerability + gadgets + some unrandomized code

- History:
  - No code randomized: Code injection
  - DEP enabled by default: ROP attacks using libc gadgets publicized ~2007
  - Libc randomized
  - ASLR library load points
  - Q builds ROP compiler using .text section
  - Today: Windows 7 compiler randomizes text by default, Randomizing text on Linux not straight-forward.

Gadgets

`Mem[v2] = v1`

Desired Logic



eax	v_1
ebx	
eip	a_1

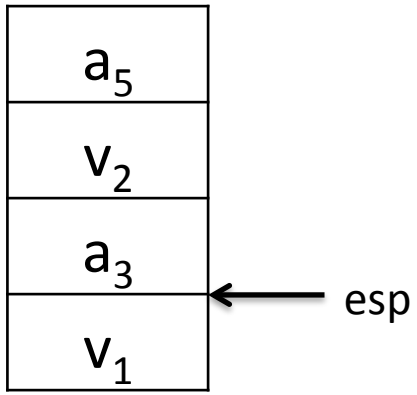
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic



Stack

eax	v_1
ebx	
eip	a_3

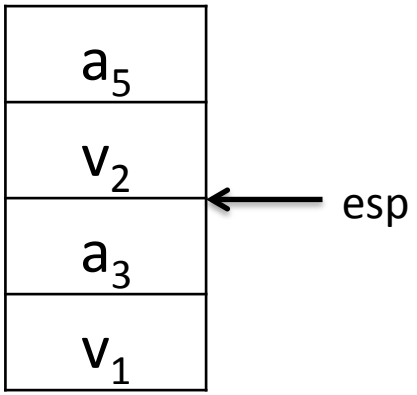
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_3

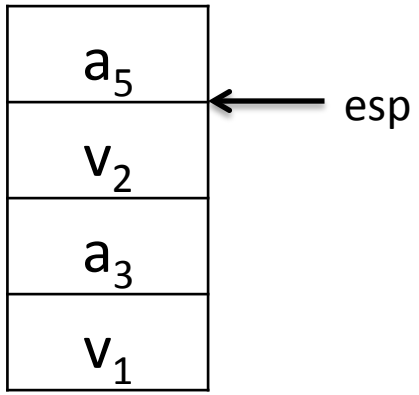
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_4

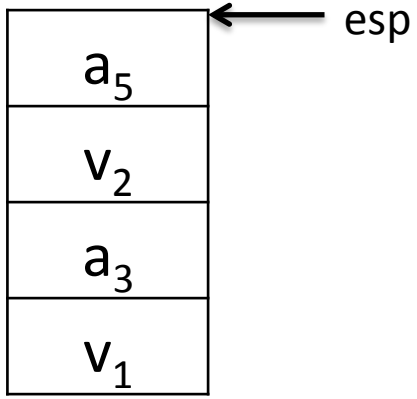
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_5

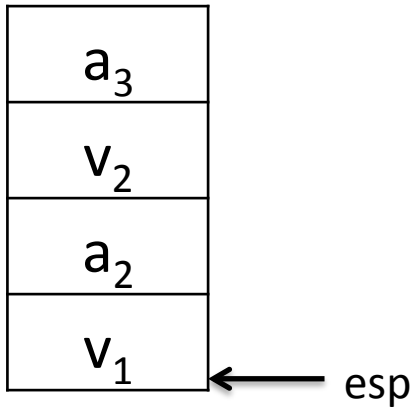
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Equivalence

```
Mem[v2] = v1
```

Desired Logic



Stack

semantically equivalent

"Gadgets"

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax`

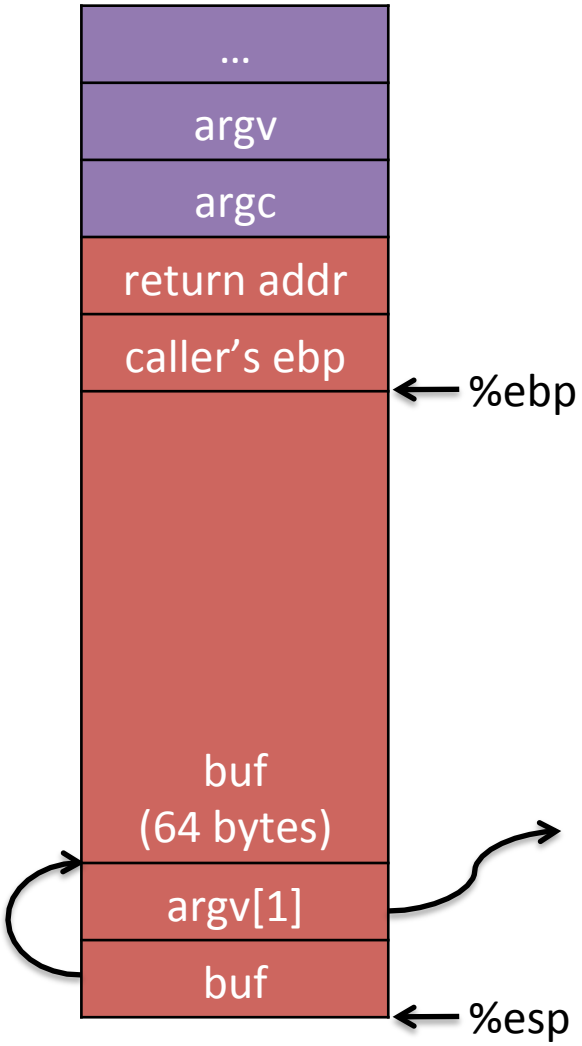
Implementation 2

Return-Oriented Programming (ROP)

```
Mem[v2] = v1
```

Desired *Shellcode*

- Find needed instruction gadgets at addresses a_1, a_2, and a_3 in *existing* code
- Overwrite stack to execute a_1, a_2, and then a_3



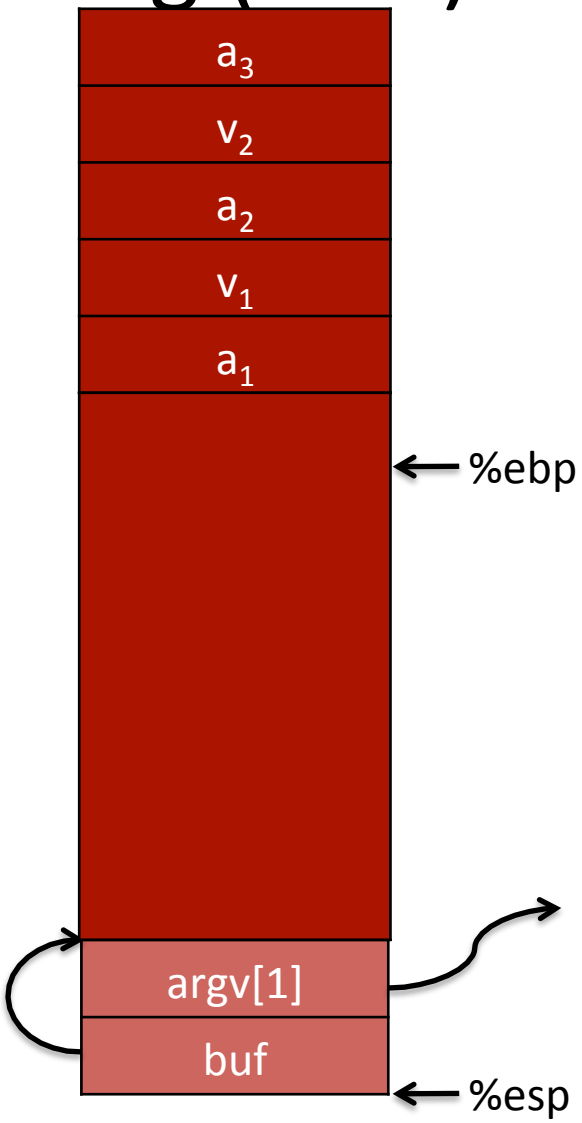
Return-Oriented Programming (ROP)

`Mem[v2] = v1`

Desired *Shellcode*

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

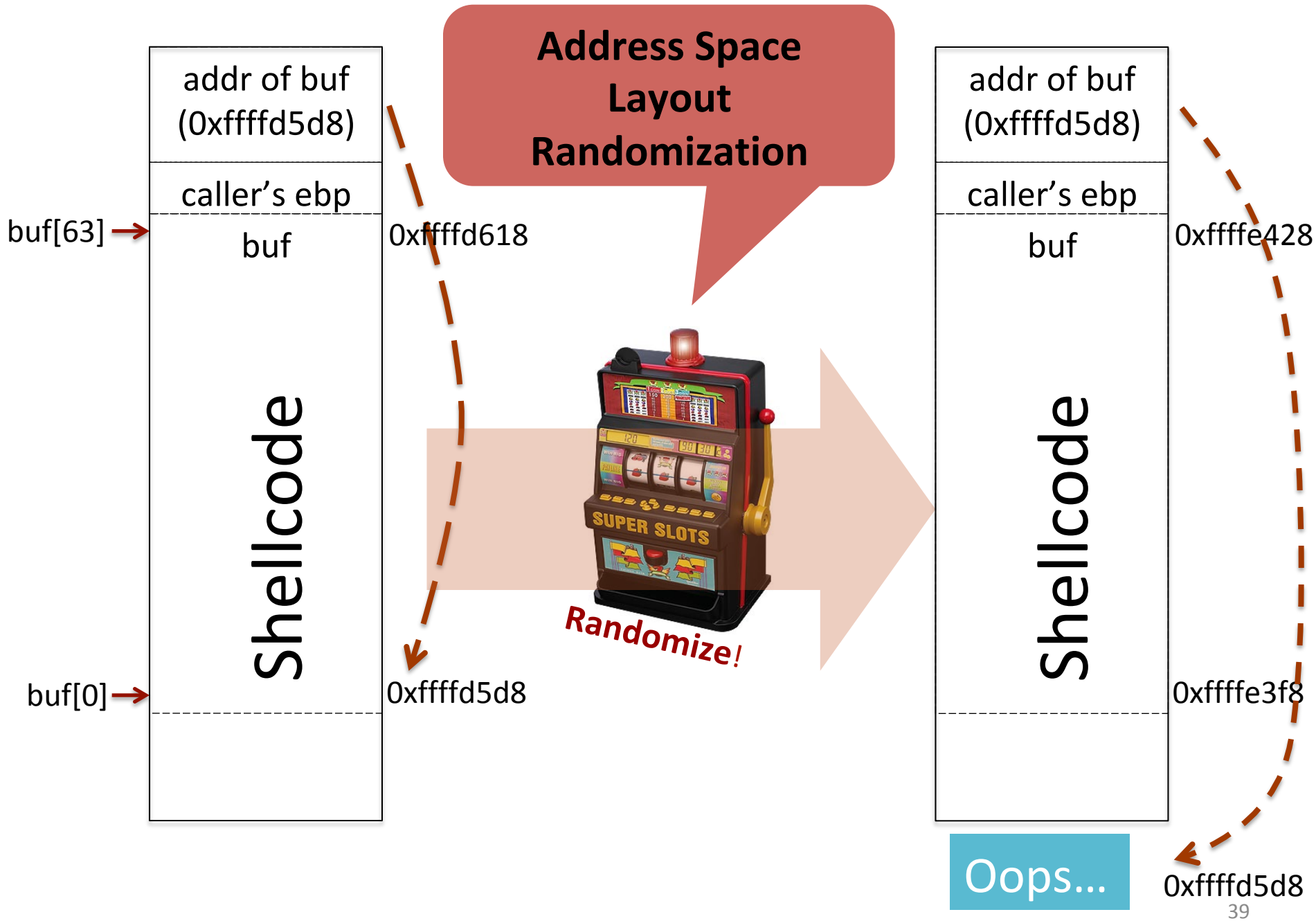
Desired store executed!



ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)

Assigned Reading:

ASLR Smack and Laugh Reference
by Tilo Muller


Address Space Layout Randomization
addr of buf (0xffffd5d8)
caller's ebp
buf
Shellcode
buf[63]
buf[0]
0xffffd618
0xffffd5d8
SUPER SLOTS
Randomize!
addr of buf (0xffffd5d8)
caller's ebp
buf
Shellcode
0xffffe428
0xffffe3f8
Oops...
0xffffd5d8

ASLR

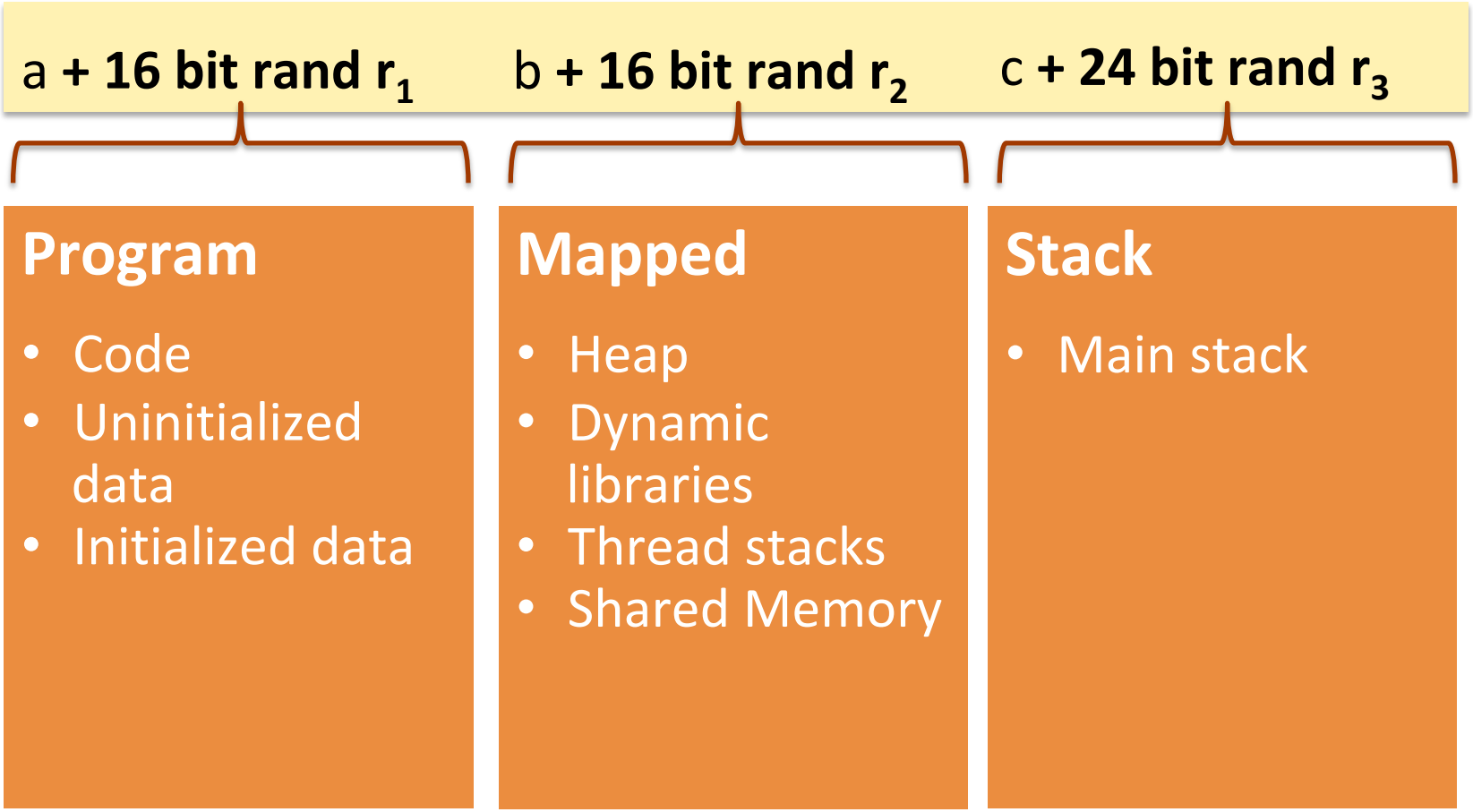
Traditional exploits need precise addresses

- *stack-based overflows:* location of shell code
- *return-to-libc:* library addresses

- **Problem:** program's memory layout is fixed
  - stack, heap, libraries etc.

- **Solution:** randomize addresses of each region!

ASLR Randomization



* ≈ 16 bit random number of 32-bit system. More on 64-bit systems.

Running `cat` Twice

- Run 1

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000 rw-p 082ac000 00:00 0          [heap]
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf966000-bf97b000 rw-p bffeb000 00:00 0          [stack]
```

- Run 2

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
086e8000-08709000 rw-p 086e8000 00:00 0          [heap]
b7d9a000-b7eef000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7eef000-b7ef0000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7ef0000-b7ef2000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf902000-bf917000 rw-p bffeb000 00:00 0          [stack]
```

Taxonomy of Vulnerabilities

- Buffer Overflow
- Command Injection
- Cross-Site Scripting
- Format String
- Illegal/Dangling Pointer
- Integer Overflow
- Path Manipulation
- Resource Injection
- String Termination Error
- Unsafe Reflection
- Insecure Temp File
- Double free
- Use-after-Free
- Memory Leak
- Debug Code Enabled
- Deadlock
- Race Conditions
- String Formatting Error

Integer Vulnerabilities

- Overflow – integer operations produce a value that is out of range
- Truncation – a value is stored in a type that is too small to represent the result
- Sign error – the sign bit is misinterpreted

Example

```
int len, error;
...
error = copyin((caddr_t)uap->alen,
               (caddr_t)&len, sizeof (len));
if (error) {
    fdrop(fp, p);
    return (error);
}
...
len = MIN(len, sa->sa_len);
error = copyout(sa, (caddr_t)uap->asa, (u_int)len);
```

len=-1=0xFFFFFFFF

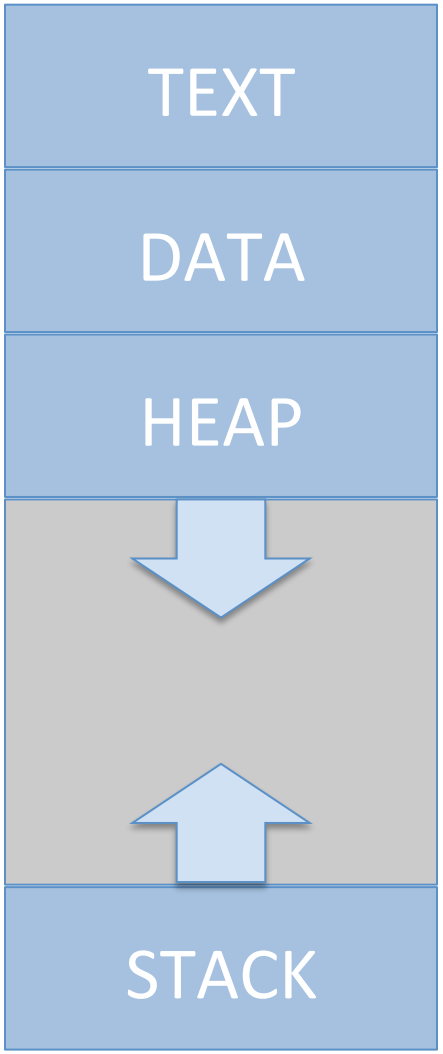
(u_int)0xFFFFFFFF=4,294,967,295

Use-After-Free (Dangling Pointer)

```
Foo *f=new Foo();
...
delete f;
...
f->bar();
```

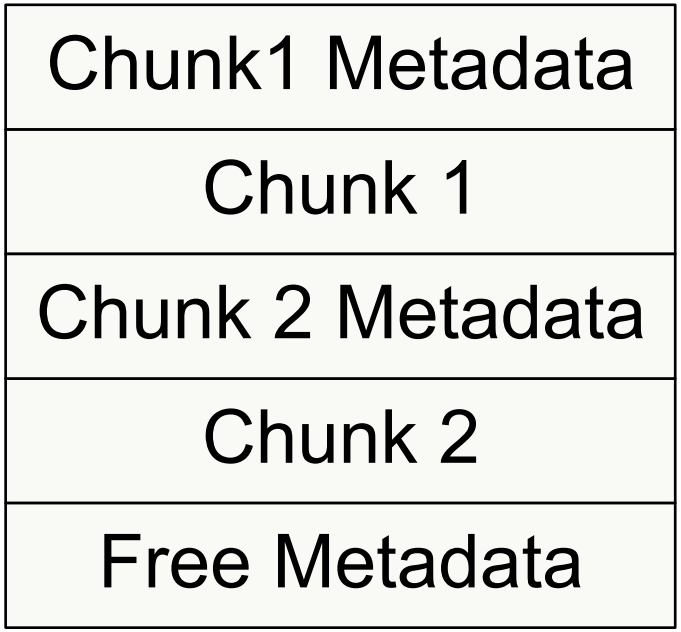
Attacking the Heap

- Allocated at run time
- Dynamic structures, objects
- Allocated in chunks by malloc
- Chunks deallocated by free
- Details are implementation specific



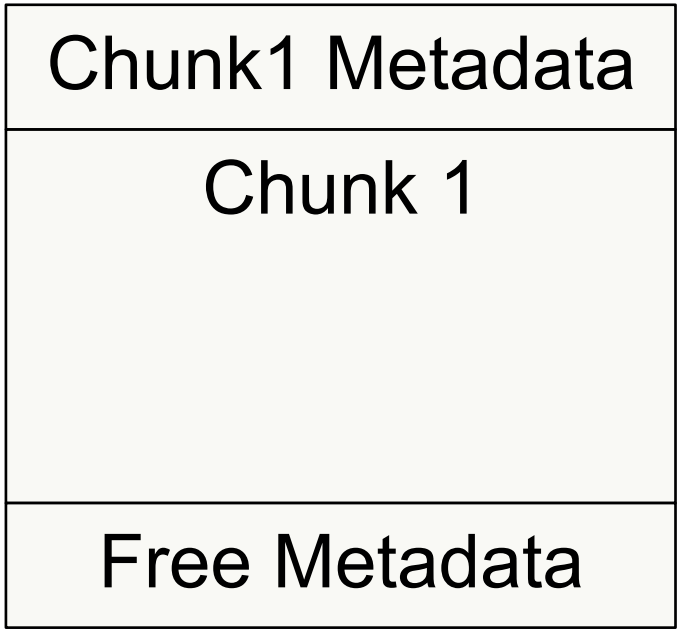
Heap Overflow

- Heap memory managed in chunks
- Chunks can grow or shrink as needed
- Chunks stored in a doubly linked list with metadata
- Chunks are allocated next to each other in memory
- Chunks can be allocated or unallocated (marked in metadata)



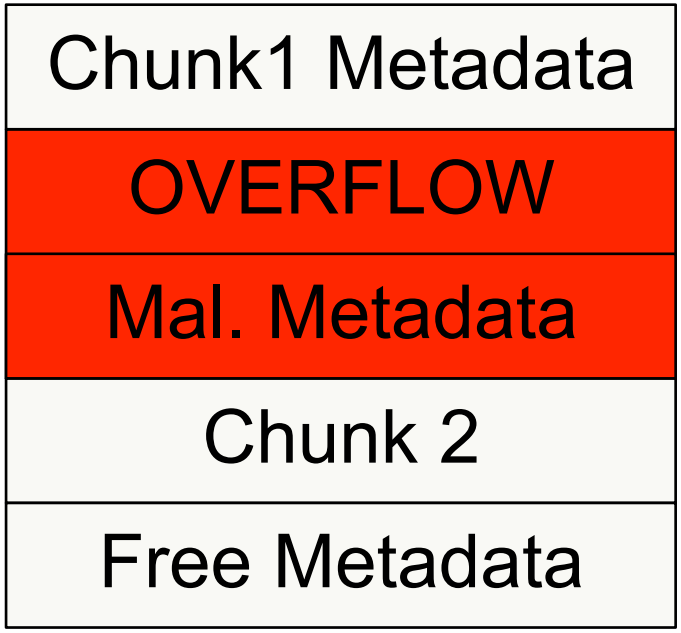
Heap Overflow

- When deallocated, a chunk is merged with its previous neighbor (if neighbor is unallocated)
- This causes a doubly linked list node to be deleted
- Deletion code *relies on metadata*!



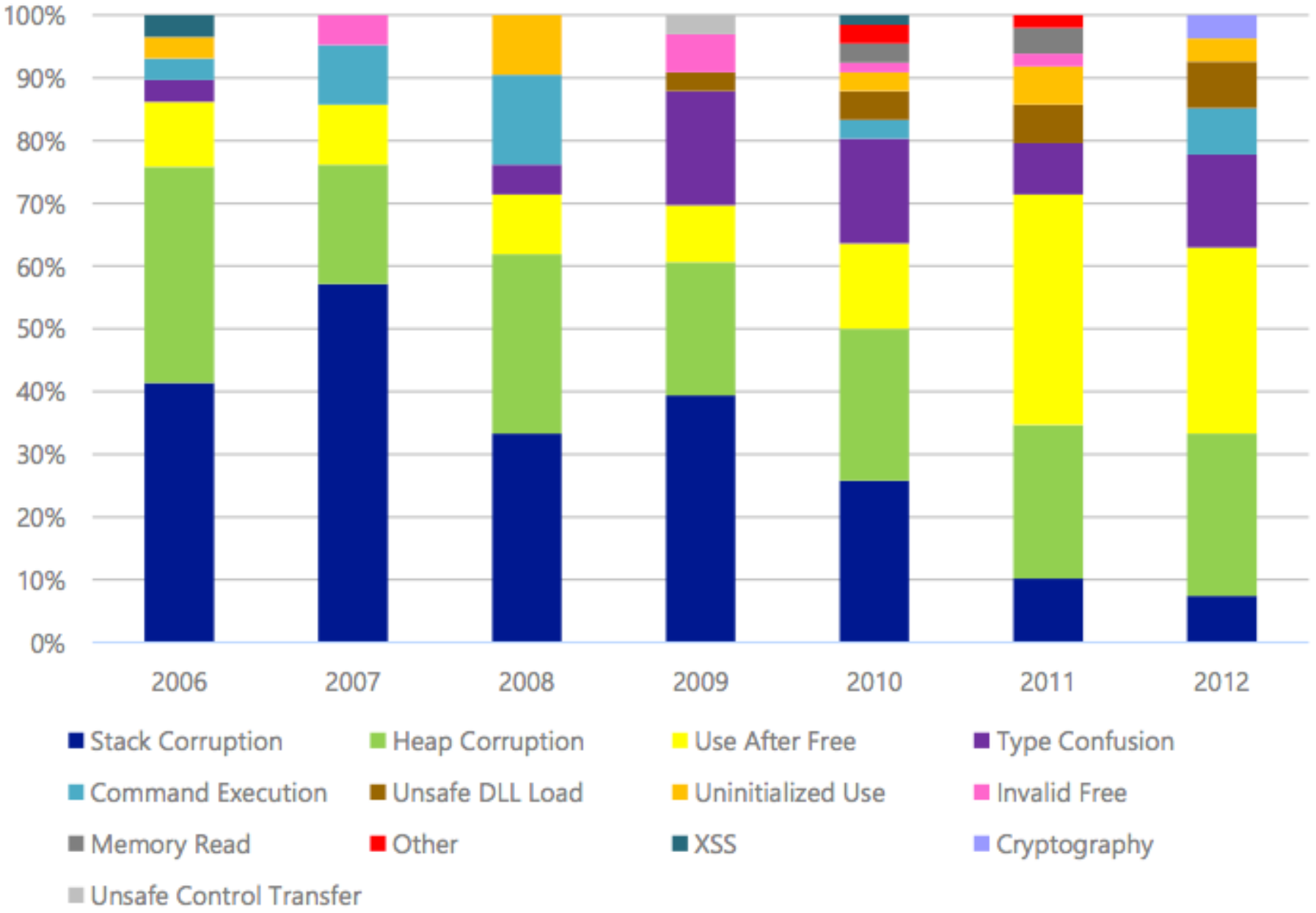
Heap Overflow

- Overflow into metadata allows attacker to control contents
- Attacker can control *what* is written and *where* it is written during node deletion



Other Heap Techniques

- Overwrite virtual method table entries
  - function pointers on heap
  - use after free
- Heap spray
  - build giant noop sleds ending in shellcode
  - spray through the entire stack
  - jumping into heap => exploit



Microsoft – Software Vulnerability Exploitation Trends 2013

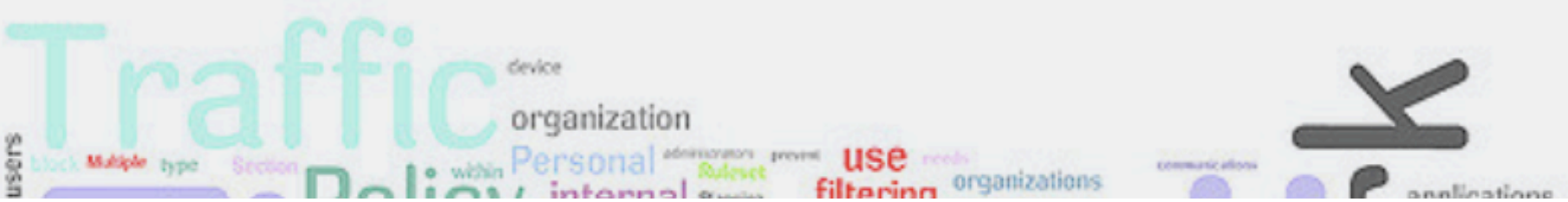
Figure 5. The distribution of CVE vulnerability classes for CVEs that are known to have been exploited

RISK ASSESSMENT —

Cisco confirms NSA-linked zeroday targeted its firewalls for years

Company advisories further corroborate authenticity of mysterious Shadow Brokers leak.

DAN GOODIN - 8/17/2016, 5:35 PM



The vulnerability is due to a buffer overflow in the affected code area.

References/Acknowledgements

- https://users.ece.cmu.edu/~dbrumley/courses/18487-f14/www/powerpoint/
- Aleph One's "Smashing the Stack for Fun and Profit" http://insecure.org/stf/smashstack.html
- Paul Makowski's "Smashing the Stack in 2011" http://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/
- Blexim's "Basic Integer Overflows" http://www.phrack.org/issues.html?issue=60&id=10
- Return-to-libc demo http://www.securitytube.net/video/258
- https://cwe.mitre.org/documents/sources/SevenPerniciousKingdoms.pdf
- http://seclists.org/bugtraq/1997/Aug/63
- https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf
- https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf
- http://phrack.org/issues/56/5.html
- http://security.stackexchange.com/questions/20497/stack-overflows-defeating-canaries-aslr-dep-nx
- http://phrack.org/issues/58/4.html
- https://cseweb.ucsd.edu/~hovav/dist/rop.pdf