`pwnlib.shellcraft.i386` — Shellcode for Intel 80386

`pwnlib.shellcraft.i386`

Shellcraft module containing generic Intel i386 shellcodes.

`pwnlib.shellcraft.i386.breakpoint`() [source]

A single-byte breakpoint instruction.

`pwnlib.shellcraft.i386.crash`() [source]

Crash.

Example

```
>>> run_assembly(shellcraft.crash()).poll(True)
-11
```

`pwnlib.shellcraft.i386.epilog`(*nargs=0*) [source]

Function epilogue.

Parameters: **nargs** (*int*) – Number of arguments to pop off the stack.

`pwnlib.shellcraft.i386.function`(*name, template_function, *registers*) [source]

Converts a shellcraft template into a callable function.

Parameters:
- **template_sz** (*callable*) – Rendered shellcode template. Any variable Arguments should be supplied as registers.
- **name** (*str*) – Name of the function.
- **registers** (*list*) – List of registers which should be filled from the stack.

```
>>> shellcode = ''
>>> shellcode += shellcraft.function('write', shellcraft.i386.linux.write, )

>>> hello = shellcraft.i386.linux.echo("Hello!", 'eax')
>>> hello_fn = shellcraft.i386.function(hello, 'eax').strip()
>>> exit = shellcraft.i386.linux.exit('edi')
>>> exit_fn = shellcraft.i386.function(exit, 'edi').strip()
>>> shellcode = '''
...     push STDOUT_FILENO
...     call hello
...     push 33
...     call exit
... hello:
...     %(hello_fn)s
... exit:
...     %(exit_fn)s
... ''' % (locals())
>>> p = run_assembly(shellcode)
>>> p.recvall()
'Hello!'
>>> p.wait_for_close()
>>> p.poll()
33
```

Notes

Can only be used on a shellcraft template which takes all of its arguments as registers. For example, the pushstr

pwnlib.shellcraft.i386.getpc(*register='ecx'*) [source]

Retrieves the value of EIP, stores it in the desired register.

Parameters: **return_value** – Value to return

pwnlib.shellcraft.i386.infloop() [source]

A two-byte infinite loop.

pwnlib.shellcraft.i386.itoa(*v, buffer='esp', allocate_stack=True*) [source]

Converts an integer into its string representation, and pushes it onto the stack.

Parameters:
- **v** (*str, int*) – Integer constant or register that contains the value to convert.
- **alloca** –

Example

```
>>> sc = shellcraft.i386.mov('eax', 0xdeadbeef)
>>> sc += shellcraft.i386.itoa('eax')
>>> sc += shellcraft.i386.linux.write(1, 'esp', 32)
>>> run_assembly(sc).recvuntil('\x00')
'3735928559\x00'
```

pwnlib.shellcraft.i386.memcpy(*dest, src, n*) [source]

Copies memory.

Parameters:

- **dest** – Destination address
- **src** – Source address
- **n** – Number of bytes

pwnlib.shellcraft.i386.mov(*dest, src, stack_allowed=True*) [source]

Move src into dest without newlines and null bytes.

If the src is a register smaller than the dest, then it will be zero-extended to fit inside the larger register.

If the src is a register larger than the dest, then only some of the bits will be used.

If src is a string that is not a register, then it will locally set *context.arch* to *'i386'* and use `pwnlib.constants.eval()` to evaluate the string. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Parameters:

- **dest** (*str*) – The destination register.
- **src** (*str*) – Either the input register, or an immediate value.
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print shellcraft.i386.mov('eax','ebx').rstrip()
    mov eax, ebx
>>> print shellcraft.i386.mov('eax', 0).rstrip()
    xor eax, eax
>>> print shellcraft.i386.mov('ax', 0).rstrip()
    xor ax, ax
>>> print shellcraft.i386.mov('ax', 17).rstrip()
    xor ax, ax
    mov al, 0x11
>>> print shellcraft.i386.mov('edi', ord('\n')).rstrip()
    push 9 /* mov edi, '\n' */
    pop edi
    inc edi
>>> print shellcraft.i386.mov('al', 'ax').rstrip()
    /* moving ax into al, but this is a no-op */
>>> print shellcraft.i386.mov('al','ax').rstrip()
    /* moving ax into al, but this is a no-op */
>>> print shellcraft.i386.mov('esp', 'esp').rstrip()
    /* moving esp into esp, but this is a no-op */
>>> print shellcraft.i386.mov('ax', 'bl').rstrip()
    movzx ax, bl
>>> print shellcraft.i386.mov('eax', 1).rstrip()
    push 1
    pop eax
>>> print shellcraft.i386.mov('eax', 1, stack_allowed=False).rstrip()
    xor eax, eax
    mov al, 1
>>> print shellcraft.i386.mov('eax', 0xdead00ff).rstrip()
    mov eax, -0xdead00ff
    neg eax
>>> print shellcraft.i386.mov('eax', 0xc0).rstrip()
    xor eax, eax
    mov al, 0xc0
>>> print shellcraft.i386.mov('edi', 0xc0).rstrip()
    mov edi, -0xc0
    neg edi
>>> print shellcraft.i386.mov('eax', 0xc000).rstrip()
    xor eax, eax
    mov ah, 0xc000 >> 8
>>> print shellcraft.i386.mov('eax', 0xffc000).rstrip()
    mov eax, 0x1010101
    xor eax, 0x1010101 ^ 0xffc000
>>> print shellcraft.i386.mov('edi', 0xc000).rstrip()
    mov edi, (-1) ^ 0xc000
    not edi
>>> print shellcraft.i386.mov('edi', 0xf500).rstrip()
    mov edi, 0x1010101
    xor edi, 0x1010101 ^ 0xf500
>>> print shellcraft.i386.mov('eax', 0xc0c0).rstrip()
    xor eax, eax
    mov ax, 0xc0c0
>>> print shellcraft.i386.mov('eax', 'SYS_execve').rstrip()
    push SYS_execve /* 0xb */
    pop eax
>>> with context.local(os='freebsd'):
...     print shellcraft.i386.mov('eax', 'SYS_execve').rstrip()
    push SYS_execve /* 0x3b */
    pop eax
>>> print shellcraft.i386.mov('eax', 'PROT_READ | PROT_WRITE | PROT_EXEC').rstrip()
    push (PROT_READ | PROT_WRITE | PROT_EXEC) /* 7 */
    pop eax
```

pwnlib.shellcraft.i386.nop() [source]

A single-byte nop instruction.

pwnlib.shellcraft.i386.prolog() [source]

Function prologue.

pwnlib.shellcraft.i386.push(*value*) [source]

Pushes a value onto the stack without using null bytes or newline characters.

If src is a string, then we try to evaluate with *context.arch* = *'i386'* using `pwnlib.constants.eval()` before determining how to push it. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Parameters: **value** (*int*,*str*) – The value or register to push

Example

```
>>> print pwnlib.shellcraft.i386.push(0).rstrip()
    /* push 0 */
    push 1
    dec byte ptr [esp]
>>> print pwnlib.shellcraft.i386.push(1).rstrip()
    /* push 1 */
    push 1
>>> print pwnlib.shellcraft.i386.push(256).rstrip()
    /* push 0x100 */
    push 0x1010201
    xor dword ptr [esp], 0x1010301
>>> print pwnlib.shellcraft.i386.push('SYS_execve').rstrip()
    /* push SYS_execve (0xb) */
    push 0xb
>>> print pwnlib.shellcraft.i386.push('SYS_sendfile').rstrip()
    /* push SYS_sendfile (0xbb) */
    push 0x1010101
    xor dword ptr [esp], 0x10101ba
>>> with context.local(os = 'freebsd'):
...     print pwnlib.shellcraft.i386.push('SYS_execve').rstrip()
    /* push SYS_execve (0x3b) */
    push 0x3b
```

pwnlib.shellcraft.i386.pushstr(*string*, *append_null=True*) [source]

Pushes a string onto the stack without using null bytes or newline characters.

Example

```
>>> print shellcraft.i386.pushstr('').rstrip()
    /* push '\x00' */
    push 1
    dec byte ptr [esp]
>>> print shellcraft.i386.pushstr('a').rstrip()
    /* push 'a\x00' */
    push 0x61
>>> print shellcraft.i386.pushstr('aa').rstrip()
    /* push 'aa\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016060
>>> print shellcraft.i386.pushstr('aaa').rstrip()
    /* push 'aaa\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1606060
>>> print shellcraft.i386.pushstr('aaaa').rstrip()
    /* push 'aaaa\x00' */
    push 1
    dec byte ptr [esp]
    push 0x61616161
>>> print shellcraft.i386.pushstr('aaaaa').rstrip()
    /* push 'aaaaa\x00' */
    push 0x61
    push 0x61616161
>>> print shellcraft.i386.pushstr('aaaa', append_null = False).rstrip()
    /* push 'aaaa' */
    push 0x61616161
>>> print shellcraft.i386.pushstr('\xc3').rstrip()
    /* push '\xc3\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x10101c2
>>> print shellcraft.i386.pushstr('\xc3', append_null = False).rstrip()
    /* push '\xc3' */
    push -0x3d
>>> with context.local():
...    context.arch = 'i386'
...    print enhex(asm(shellcraft.pushstr("/bin/sh")))
68010101018134242e726901682f62696e
>>> with context.local():
...    context.arch = 'i386'
...    print enhex(asm(shellcraft.pushstr("")))
6a01fe0c24
>>> with context.local():
...    context.arch = 'i386'
...    print enhex(asm(shellcraft.pushstr("\x00", False)))
6a01fe0c24
```

Parameters:

- **string** (*str*) – The string to push.
- **append_null** (*bool*) – Whether to append a single NULL-byte before pushing.

`pwnlib.shellcraft.i386.pushstr_array`(*reg, array*) [source]

Pushes an array/envp-style array of pointers onto the stack.

Parameters:

- **reg** (*str*) – Destination register to hold the pointer.
- **array** (*str*,*list*) – Single argument or list of arguments to push. NULL termination is normalized so that each argument ends with exactly one NULL byte.

pwnlib.shellcraft.i386.ret(*return_value=None*) [source]

A single-byte RET instruction.

Parameters: **return_value** – Value to return

pwnlib.shellcraft.i386.setregs(*reg_context, stack_allowed=True*) [source]

Sets multiple registers, taking any register dependencies into account (i.e., given eax=1,ebx=eax, set ebx first).

Parameters:
- **reg_context** (*dict*) – Desired register context
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print shellcraft.setregs({'eax':1, 'ebx':'eax'}).rstrip()
    mov ebx, eax
    push 1
    pop eax
>>> print shellcraft.setregs({'eax':'ebx', 'ebx':'eax', 'ecx':'ebx'}).rstrip()
    mov ecx, ebx
    xchg eax, ebx
```

pwnlib.shellcraft.i386.stackarg(*index, register*) [source]

Loads a stack-based argument into a register.

Assumes that the 'prolog' code was used to save EBP.

Parameters:
- **index** (*int*) – Zero-based argument index.
- **register** (*str*) – Register name.

pwnlib.shellcraft.i386.stackhunter(*cookie = 0x7afceb58*) [source]

Returns an an egghunter, which searches from esp and upwards for a cookie. However to save bytes, it only looks at a single 4-byte alignment. Use the function stackhunter_helper to generate a suitable cookie prefix for you.

The default cookie has been chosen, because it makes it possible to shave a single byte, but other cookies can be used too.

Example

```
>>> with context.local():
...     context.arch = 'i386'
...     print enhex(asm(shellcraft.stackhunter()))
3d58ebfc7a75faffe4
>>> with context.local():
...     context.arch = 'i386'
...     print enhex(asm(shellcraft.stackhunter(0xdeadbeef)))
583defbeadde75f8ffe4
```

pwnlib.shellcraft.i386.strcpy(*dst*, *src*) [source]

Copies a string

Example

```
>>> sc  = 'jmp get_str\n'
>>> sc += 'pop_str: pop eax\n'
>>> sc += shellcraft.i386.strcpy('esp', 'eax')
>>> sc += shellcraft.i386.linux.write(1, 'esp', 32)
>>> sc += shellcraft.i386.linux.exit(0)
>>> sc += 'get_str: call pop_str\n'
>>> sc += '.asciz "Hello, world\\n"'
>>> run_assembly(sc).recvline()
'Hello, world\n'
```

pwnlib.shellcraft.i386.strlen(*string*, *reg='ecx'*) [source]

Calculate the length of the specified string.

Parameters:
- **string** (*str*) – Register or address with the string
- **reg** (*str*) – Named register to return the value in, ecx is the default.

Example

```
>>> sc  = 'jmp get_str\n'
>>> sc += 'pop_str: pop eax\n'
>>> sc += shellcraft.i386.strlen('eax')
>>> sc += 'push ecx;'
>>> sc += shellcraft.i386.linux.write(1, 'esp', 4)
>>> sc += shellcraft.i386.linux.exit(0)
>>> sc += 'get_str: call pop_str\n'
>>> sc += '.asciz "Hello, world\\n"'
>>> run_assembly(sc).unpack() == len('Hello, world\n')
True
```

pwnlib.shellcraft.i386.trap() [source]

A trap instruction.

pwnlib.shellcraft.i386.xor(*key*, *address*, *count*) [source]

XORs data a constant value.

Parameters:

- **key** (*int*,*str*) – XOR key either as a 4-byte integer, If a string, length must be a power of two, and not longer than 4 bytes. Alternately, may be a register.
- **address** (*int*) – Address of the data (e.g. 0xdead0000, 'esp')
- **count** (*int*) – Number of bytes to XOR, or a register containing the number of bytes to XOR.

Example

```
>>> sc  = shellcraft.read(0, 'esp', 32)
>>> sc += shellcraft.xor(0xdeadbeef, 'esp', 32)
>>> sc += shellcraft.write(1, 'esp', 32)
>>> io = run_assembly(sc)
>>> io.send(cyclic(32))
>>> result = io.recvn(32)
>>> expected = xor(cyclic(32), p32(0xdeadbeef))
>>> result == expected
True
```

`pwnlib.shellcraft.i386.linux`

Shellcraft module containing Intel i386 shellcodes for Linux.

`pwnlib.shellcraft.i386.linux.acceptloop_ipv4`(*port*) [source]

Parameters: **port** (*int*) – the listening port

Waits for a connection. Leaves socket in EBP. ipv4 only

`pwnlib.shellcraft.i386.linux.cat`(*filename, fd=1*) [source]

Opens a file and writes its contents to the specified file descriptor.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG')
>>> run_assembly(shellcraft.i386.linux.cat(f)).recvall()
'FLAG'
```

`pwnlib.shellcraft.i386.linux.connect`(*host, port, network='ipv4'*) [source]

Connects to the host on the specified port. Leaves the connected socket in edx

Parameters:

- **host** (*str*) – Remote IP address or hostname (as a dotted quad / string)
- **port** (*int*) – Remote port
- **network** (*str*) – Network protocol (ipv4 or ipv6)

Examples

```
>>> l = listen(timeout=5)
>>> assembly  = shellcraft.i386.linux.connect('localhost', l.lport)
>>> assembly += shellcraft.i386.pushstr('Hello')
>>> assembly += shellcraft.i386.linux.write('edx', 'esp', 5)
>>> p = run_assembly(assembly)
>>> l.wait_for_connection().recv()
'Hello'
```

```
>>> l = listen(fam='ipv6', timeout=5)
>>> assembly = shellcraft.i386.linux.connect('::1', l.lport, 'ipv6')
>>> p = run_assembly(assembly)
>>> assert l.wait_for_connection()
```

pwnlib.shellcraft.i386.linux.connectstager(*host, port, network='ipv4'*) [source]

connect recvsize stager :param host, where to connect to: :param port, which port to connect to: :param network, ipv4 or ipv6? (default: ipv4)

pwnlib.shellcraft.i386.linux.dir(*in_fd='ebp', size=2048, allocate_stack=True*) [source]

Reads to the stack from a directory.

Parameters:

- **in_fd** (*int/str*) – File descriptor to be read from.
- **size** (*int*) – Buffer size.
- **allocate_stack** (*bool*) – allocate 'size' bytes on the stack.

You can optioanlly shave a few bytes not allocating the stack space.

The size read is left in eax.

pwnlib.shellcraft.i386.linux.dupio(*sock='ebp'*) [source]

Args: [sock (imm/reg) = ebp] Duplicates sock to stdin, stdout and stderr

pwnlib.shellcraft.i386.linux.dupsh(*sock='ebp'*) [source]

Args: [sock (imm/reg) = ebp] Duplicates sock to stdin, stdout and stderr and spawns a shell.

pwnlib.shellcraft.i386.linux.echo(*string, sock='1'*) [source]

Writes a string to a file descriptor

Example

```
>>> run_assembly(shellcraft.echo('hello', 1)).recvall()
'hello'
```

pwnlib.shellcraft.i386.linux.egghunter(*egg*, *start_address = 0*) [source]

Searches memory for the byte sequence 'egg'.

Return value is the address immediately following the match, stored in RDI.

Parameters:	• **egg** (*str*, *int*) – String of bytes, or word-size integer to search for • **start_address** (*int*) – Where to start the search

pwnlib.shellcraft.i386.linux.findpeer(*port=None*) [source]

Args: port (defaults to any port) Finds a socket, which is connected to the specified port. Leaves socket in ESI.

pwnlib.shellcraft.i386.linux.findpeersh(*port=None*) [source]

Args: port (defaults to any) Finds an open socket which connects to a specified port, and then opens a dup2 shell on it.

pwnlib.shellcraft.i386.linux.findpeerstager(*port=None*) [source]

Findpeer recvsize stager :param port, the port given to findpeer: :type port, the port given to findpeer: defaults to any

pwnlib.shellcraft.i386.linux.forkbomb() [source]

Performs a forkbomb attack.

pwnlib.shellcraft.i386.linux.forkexit() [source]

Attempts to fork. If the fork is successful, the parent exits.

pwnlib.shellcraft.i386.linux.i386_to_amd64() [source]

Returns code to switch from i386 to amd64 mode.

pwnlib.shellcraft.i386.linux.killparent() [source]

Kills its parent process until whatever the parent is (probably init) cannot be killed any longer.

pwnlib.shellcraft.i386.linux.loader(*address*) [source]

Loads a statically-linked ELF into memory and transfers control.

Parameters:	**address** (*int*) – Address of the ELF as a register or integer.

pwnlib.shellcraft.i386.linux.loader_append(*data=None*) [source]

Loads a statically-linked ELF into memory and transfers control.

Similar to loader.asm but loads an appended ELF.

Parameters: **data** (*str*) – If a valid filename, the data is loaded from the named file. Otherwise, this is treated as raw ELF data to append. If `None`, it is ignored.

Example

```
>>> gcc = process(['gcc','-m32','-xc','-static','-Wl,-Ttext-segment=0x20000000','-
'])
>>> gcc.write('''
... int main() {
...     printf("Hello, %s!\\n", "i386");
... }
... ''')
>>> gcc.shutdown('send')
>>> gcc.poll(True)
0
>>> sc = shellcraft.loader_append('a.out')
```

The following doctest is commented out because it doesn't work on Travis for reasons I cannot diagnose. However, it should work just fine :-)

> # >>> run_assembly(sc).recvline() == 'Hello, i386!n' # True

pwnlib.shellcraft.i386.linux.mprotect_all(*clear_ebx=True, fix_null=False*) [source]

Calls mprotect(page, 4096, PROT_READ | PROT_WRITE | PROT_EXEC) for every page.

It takes around 0.3 seconds on my box, but your milage may vary.

Parameters:

- **clear_ebx** (*bool*) – If this is set to False, then the shellcode will assume that ebx has already been zeroed.
- **fix_null** (*bool*) – If this is set to True, then the NULL-page will also be mprotected at the cost of slightly larger shellcode

pwnlib.shellcraft.i386.linux.pidmax() [source]

Retrieves the highest numbered PID on the system, according to the sysctl kernel.pid_max.

pwnlib.shellcraft.i386.linux.readfile(*path, dst='esi'*) [source]

Args: [path, dst (imm/reg) = esi] Opens the specified file path and sends its content to the specified file descriptor.

pwnlib.shellcraft.i386.linux.readn(*fd, buf, nbytes*) [source]

Reads exactly nbytes bytes from file descriptor fd into the buffer buf.

Parameters:
- **fd** (*int*) – fd
- **buf** (*void*) – buf
- **nbytes** (*size_t*) – nbytes

pwnlib.shellcraft.i386.linux.recvsize(*sock, reg='ecx'*) [source]

Recives 4 bytes size field Useful in conjuncion with findpeer and stager :param sock, the socket to read the payload from.: :param reg, the place to put the size: :type reg, the place to put the size: default ecx

Leaves socket in ebx

pwnlib.shellcraft.i386.linux.setregid(*gid='egid'*) [source]

Args: [gid (imm/reg) = egid] Sets the real and effective group id.

pwnlib.shellcraft.i386.linux.setreuid(*uid='euid'*) [source]

Args: [uid (imm/reg) = euid] Sets the real and effective user id.

pwnlib.shellcraft.i386.linux.sh() [source]

Execute a different process.

```
>>> p = run_assembly(shellcraft.i386.linux.sh())
>>> p.sendline('echo Hello')
>>> p.recv()
'Hello\n'
```

pwnlib.shellcraft.i386.linux.socket(*network='ipv4', proto='tcp'*) [source]

Creates a new socket

pwnlib.shellcraft.i386.linux.socketcall(*socketcall, socket, sockaddr, sockaddr_len*) [source]

Invokes a socket call (e.g. socket, send, recv, shutdown)

pwnlib.shellcraft.i386.linux.stage(*fd=0, length=None*) [source]

Migrates shellcode to a new buffer.

Parameters:
- **fd** (*int*) – Integer file descriptor to recv data from. Default is stdin (0).
- **length** (*int*) – Optional buffer length. If None, the first pointer-width of data received is the length.

Example

```
>>> p = run_assembly(shellcraft.stage())
>>> sc = asm(shellcraft.echo("Hello\n", constants.STDOUT_FILENO))
>>> p.pack(len(sc))
>>> p.send(sc)
>>> p.recvline()
'Hello\n'
```

`pwnlib.shellcraft.i386.linux.stager`(*sock, size, handle_error=False, tiny=False*) [source]

Recives a fixed sized payload into a mmaped buffer Useful in conjuncion with findpeer. :param sock, the socket to read the payload from.: :param size, the size of the payload:

`pwnlib.shellcraft.i386.linux.syscall`(*syscall=None, arg0=None, arg1=None, arg2=None, arg3=None, arg4=None, arg5=None*) [source]

> Args: [syscall_number, *args]

Does a syscall

Any of the arguments can be expressions to be evaluated by `pwnlib.constants.eval()`.

Example

```
>>> print pwnlib.shellcraft.i386.linux.syscall('SYS_execve', 1, 'esp', 2,
0).rstrip()
    /* call execve(1, 'esp', 2, 0) */
    push SYS_execve /* 0xb */
    pop eax
    push 1
    pop ebx
    mov ecx, esp
    push 2
    pop edx
    xor esi, esi
    int 0x80
>>> print pwnlib.shellcraft.i386.linux.syscall('SYS_execve', 2, 1, 0, 20).rstrip()
    /* call execve(2, 1, 0, 0x14) */
    push SYS_execve /* 0xb */
    pop eax
    push 2
    pop ebx
    push 1
    pop ecx
    push 0x14
    pop esi
    cdq /* edx=0 */
    int 0x80
>>> print pwnlib.shellcraft.i386.linux.syscall().rstrip()
    /* call syscall() */
    int 0x80
>>> print pwnlib.shellcraft.i386.linux.syscall('eax', 'ebx', 'ecx').rstrip()
    /* call syscall('eax', 'ebx', 'ecx') */
    /* setregs noop */
    int 0x80
>>> print pwnlib.shellcraft.i386.linux.syscall('ebp', None, None, 1).rstrip()
    /* call syscall('ebp', ?, ?, 1) */
    mov eax, ebp
    push 1
    pop edx
    int 0x80
>>> print pwnlib.shellcraft.i386.linux.syscall(
...               'SYS_mmap2', 0, 0x1000,
...               'PROT_READ | PROT_WRITE | PROT_EXEC',
...               'MAP_PRIVATE | MAP_ANONYMOUS',
...               -1, 0).rstrip()
    /* call mmap2(0, 0x1000, 'PROT_READ | PROT_WRITE | PROT_EXEC', 'MAP_PRIVATE |
MAP_ANONYMOUS', -1, 0) */
    xor eax, eax
    mov al, 0xc0
    xor ebp, ebp
    xor ebx, ebx
    xor ecx, ecx
    mov ch, 0x1000 >> 8
    push -1
    pop edi
    push (PROT_READ | PROT_WRITE | PROT_EXEC) /* 7 */
    pop edx
    push (MAP_PRIVATE | MAP_ANONYMOUS) /* 0x22 */
    pop esi
    int 0x80
>>> print pwnlib.shellcraft.open('/home/pwn/flag').rstrip()
    /* open(file='/home/pwn/flag', oflag=0, mode=0) */
    /* push '/home/pwn/flag\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016660
    push 0x6c662f6e
    push 0x77702f65
    push 0x6d6f682f
    mov ebx, esp
    xor ecx, ecx
    xor edx, edx
    /* call open() */
    push SYS_open /* 5 */
    pop eax
    int 0x80
```

`pwnlib.shellcraft.i386.freebsd`

Shellcraft module containing Intel i386 shellcodes for FreeBSD.

`pwnlib.shellcraft.i386.freebsd.acceptloop_ipv4`(*port*) [source]

Args: port Waits for a connection. Leaves socket in EBP. ipv4 only

`pwnlib.shellcraft.i386.freebsd.i386_to_amd64`() [source]

Returns code to switch from i386 to amd64 mode.

`pwnlib.shellcraft.i386.freebsd.sh`() [source]

Execute /bin/sh